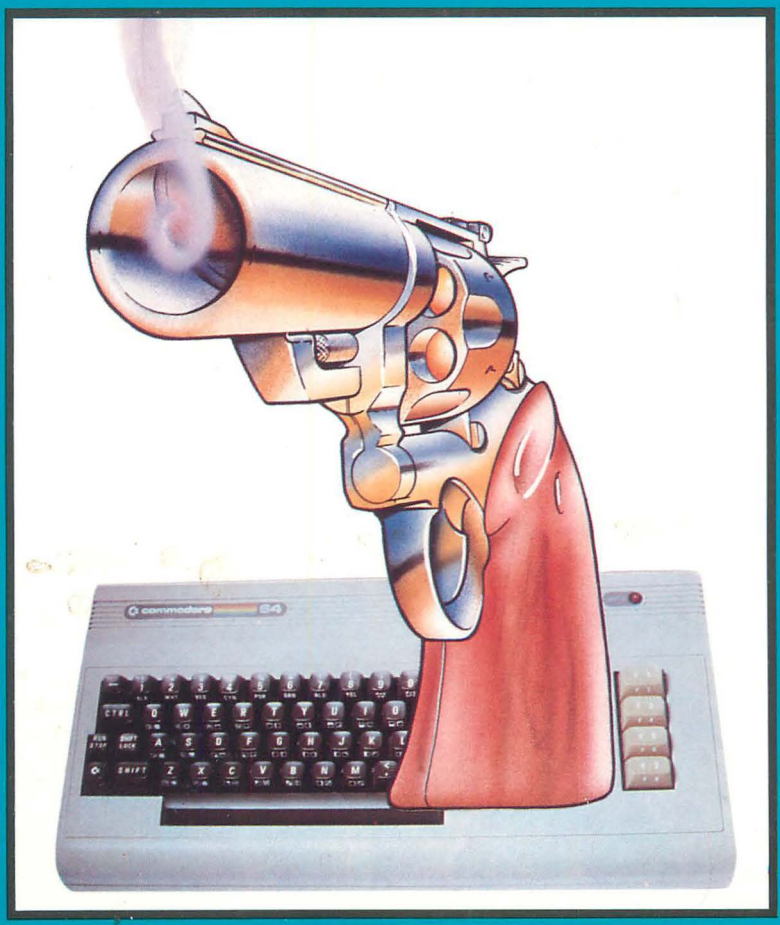
CENTURY COMMUNICATIONS

THE PROGRAMMERS' TROUBLE SHOOTING GUIDE

COMMODORE 64



PIERS LETCHER

THE PROGRAMMERS' TROUBLESHOOTING GUIDE

COMMODORE 64

Piers Letcher

Century Communications

London

ABOUT THE AUTHOR

After graduating in computer systems, Piers Letcher worked in many areas of the computer world, from programming and maintaining mainframes, to designing educational software for home computers. He has contributed to many computer journals, including *Input* and *Your Computer*, and has also worked as Peripherals Editor on the weekly magazine *Personal Computer News*. He has written several books about microcomputers, including one on peripherals and two on graphics. His other interests include English humour and foreign mountains.

First published in Great Britain in 1985 by
Century Communications Limited,
part of the Century – Hutchinson group
12 – 13 Greek Street, London, W1V 5LE

ISBN 0 7126 0600 9

Originated directly from the author's
word-processor disks by
NWL Editorial Services, Langport, Somerset

Printed in Great Britain by Billing & Sons Ltd., Worcester

To Martine

CONTENTS

PREFACE

Anyone using a computer for the first time has to learn the majority of the problems, and solutions, of programming without outside help. This process can take years and to a large extent is wasted time, since many of the problems and solutions are already known by people who have been programming for some time. Some of the problems you will find are just in the nature of computers, but many are due to the particular nature of the Commodore 64.

This book should help you to bypass much of the gradual learning process, and at the same time to become a better and more efficient programmer. If you use this book constructively it could save you alot of time, effort and frustration. Error-trapping and debugging are both wearisome and time-wasting pursuits; any time saved can be put to much better use writing new programs.

A number of people were influential and helpful in the writing of this book. In particular my thanks go to Simon Dally at *Century Communications* for commissioning the book, and to Jacqui Lyons for her excellent representation. Thanks also to my parents and brother for their generosity with food and a quiet place to write, to Bonzo for getting me here, and to Bingy for his company.

Piers Letcher
October 1984

1

INTRODUCTION

You will probably have discovered, within hours of setting up your Commodore 64 for the first time, that you can encounter an enormous range of software problems. Some of these are easy to solve, and some are not so easy, but one thing is certain — solving these problems will take up more of your time than programming ever will. This applies to all programmers, from those who are writing a program for the first time to those who make a living from it.

ABOUT THIS BOOK

What this book aims to do is to show you where many of the problems lie, how to isolate them and, most of all, how to cure them. Every type of problem you are likely to face is covered, and the book has been split into sections which deal with the areas in which errors are most commonly found. Extra chapters cover the Commodore 64's memory, program design and structure, and machine code programming. With all this you should be well equipped not just to troubleshoot on the Commodore 64, but to take that knowledge with you, when you move on to other computers in the future.

WHAT IS TROUBLESHOOTING?

One of the first questions you probably wanted answered when you saw this book was 'What exactly is troubleshooting?' Program troubleshooting is the process you go through to eliminate errors from a program. The first stage of this process must be detection of the error, which is not the same as knowing that you have an error — that much should be clear already. Detecting the error relies on a series of steps you take to find out precisely where the problem lies. Mostly, this is not as easy as it sounds, since quite often you will not know an error even if you are looking straight at it. In other cases, the error may not be in the place that appears to be causing the trouble. You frequently get errors that do not show up until some secondary effect is noticed, and the book details such cases, as well as the direct errors in the current problem statement or line.

This book will help you spot just where the errors are, and, having done this, it will help you to solve the problems. Many of these are simple to cure once you know what is causing the error, but there are others which are insoluble, and the book is careful to pinpoint these. There is no point in battling on against an error if there is no simple cure for it. Instead it is better to abandon that line of thinking and rewrite that section of the program in a different way.

HOW THIS BOOK IS BROKEN DOWN

The book starts with a look at the problems of getting programs into the computer in the first place, since this is the area where many of the primary errors can creep in, whether it is due to mistyping, or a mistaken use of the Commodore's keyboard, with all its potential problems. Chapters 3 and 4 deal with the problems inherent in the BASIC language: Chapter 3 looks at BASIC Run errors and the best way of trapping them, while Chapter 4 goes on to look at specific areas of Commodore BASIC which cause trouble. This is the longest chapter covering all the different problems that occur with the various BASIC constructs. It is here that you will find details of many of the BASIC keywords.

Chapter 5 deals with input. Although it may seem surprising, most computers are not naturally equipped with input and output facilities. This can make life difficult for the user of some machines, but fortunately most home computers have special input and output routines built into the BASIC language. The Commodore 64 has several commands in BASIC to facilitate the transfer of information to and from the computer, and this chapter looks at the typical problems you might find here, and how best to avoid them. Topics covered include getting data from cassette or disk drive, from the user port and joysticks and from the keyboard, the latter using GET and INPUT. OPEN, CLOSE, CMD, INPUT#, GET#, READ, DATA and RESTORE are also covered.

Chapter 6 looks at output. Many of the same problems applying to input also apply to output — though they are, of course, opposite. Typical problems that might be encountered while saving to cassette or disk, or sending information to any other output device (printer etc) are examined. Also looked at in detail is the most important output that you will need for most programming: output to the screen. This can be done using the BASIC keywords PRINT and POKE, but each has its own associated problems.

Graphics is the topic covered in Chapter 7. This is one of the most interesting areas to work in, but because it involves the placing of objects and drawing on the screen it is often difficult to achieve the exact effect you are looking for. This chapter discusses the individual problems associated with each of the main areas of stationary graphics (movement is covered in the next chapter). Colours and high resolution are examined in detail, along with the block graphics and more sophisticated methods of making pictures.

Chapter 8 examines movement or animated graphics. Creating movement is one of the most satisfying ways of using your Commodore 64, but the use of sprites is rather complex. The problem lies partly in the sheer flexibility of the sprite hardware provided — to get the best movement you must first have the sprites correctly programmed. Machine code movement and animation are also covered, along with the general positioning of objects on the screen. Pointers to the places from which problems arise are provided, building on the ideas from the previous chapter.

Chapter 9 discusses sound — one of the hardest areas of Commodore programming. The sound chip (SID) is highly complex and flexible, and because you can do so much with it it is often hard to know where to start and where to find errors when the sound produced is not the sound you wanted. Another problem with sound is that most of us are not musical enough to exploit this flexibility.

Other problems with music are often related to timing, and this can change if the program is changed, or if a lot happens between the playing of notes. Comments are given in this chapter about how to cure any problems you come across, as well as how to make life easier when using sound effects.

Chapter 10 explores memory which is the real heart of the Commodore 64. The way in which memory is used will have a considerable effect on the quality of your programs. However, once you start to delve into memory you are bound to encounter problems, especially if you are using machine code. Equally, if you want to store lots of graphics, or even complete screens, there is room to do this, but it is not always easy to find out just where. The chapter also looks at the problems you are likely to encounter if you delve too deep, or if you try to meddle with the operating system or system variables.

Throughout the book a certain emphasis has been placed on tidy programming, simply because this makes your work easier to debug. An explanation of the reasons behind this plus some of the more practical aspects of structure are given in Chapter 11 — rather than concentrating on the theoretical, which is not actually going to affect those of us who program Commodore 64s.

Unless you have access to a language like Pascal then the finer points of structure are going to be irrelevant. The importance of documentation is also mentioned, an area often ignored by programmers. It may prove to be quicker and easier not to document at a program's inception but, later on, when the errors start to appear, even the smallest amount of documentation will help enormously. This is especially true on the Commodore, where practical structuring is hard to implement due to the nature of the BASIC used.

Machine code is the subject of Chapter 12. Although really outside the scope of this book, it is interesting to see that many of the same principles apply to machine code debugging as apply to BASIC. This chapter has a look at the 6510 processor (a slight variation on the 6502) and some of the more obvious problems which can delay your progress. A description is also given of how to integrate BASIC and machine code and of the similarities of their problems. Complex topics (like animation) require a detailed knowledge of machine code beyond the scope of this book. However, the coverage has been thorough enough to ensure that when readers get as far as the subjects described, they will know the direction in which they are heading.

If you are having problems with using a particular keyword then you should refer to the appendices or the index, otherwise I would recommend reading each section as you start programming in that area.

The book ends with some useful appendices: one for anyone already familiar with computers but using a Commodore 64 for the first time; and the second explaining the Commodore 64 run messages — with hints on curing the problems which lead to them.

USING THIS BOOK FOR PROBLEM SOLVING

I recommend that you read the relevant sections and chapters before you start programming in a particular area. If you are having trouble with a specific keyword then have a look in the index to track it down. The Contents list at the front may also be helpful for specific problems, as it gives all the main sub-headings which should help you to narrow down which section of the book you might need.

Otherwise here is a list of problems, and areas in the book where a solution might be found.

Unable to run a program

Typing errors	(Chapter 2)
Corrupted memory	(Chapter 10)
Locked in machine code	(Chapter 12)

Nothing happening on the screen

Infinite loops	(Chapter 4)
Searching for a program	(Chapter 5)
Machine crash	(Chapters 10 and 12)

Problems with loading

Loading from cassette	(Chapter 5)
Loading from disk	(Chapter 5)

Problems with saving

Saving to cassette	(Chapter 6)
Saving to disk	(Chapter 6)
Verifying	(Chapter 6)

Problems with input from keyboard

Keyboard	(Chapter 5)

Problems with input from other peripherals

Peripherals	(Chapter 5)

Problems with output to screen

Screen Output	(Chapter 6)
Graphics	(Chapter 7)
Movement	(Chapter 8)
Memory	(Chapter 10)

Problems with output to other peripherals

Peripherals	(Chapter 6)

Problems with colour

Problems with drawing

Problems with redefined images

Problems with movement

Problems with sound

Problems with memory layout

Problems structuring programs

Problems starting in machine code

2

THE DIFFICULTIES OF CORRECT PROGRAM INPUT

STRUCTURE AND PROGRAM WRITING

How do *you* write a computer program? Some people will sit down with paper and pen and draw a map of the program (called a *flowchart* by those who mind about these things). Only when they are satisfied with this will they think about programming; turning each little part of the flowchart into a few lines of computer program. It will not be until the whole program has been written out on paper, together with comments about what each section does, that the computer keyboard is approached.

Others may scoff at this idea, but then proceed to do much the same thing. This group of people may be heard to use words like structure, and will replace the idea of the flowchart with that of the *block diagram*. In practice it is easier to program from a block diagram, as each block represents something that can be tested and written separately. Once you have put these blocks together you have, in theory, a complete working program.

These ideas may all seem very strange. Nowadays most people have a computer sitting in front of them, and are therefore hardly likely to resort to something as outdated as pen and paper if they have a screen and keyboard at their fingertips. You and I are probably going to dive in and start programming our Commodore 64s straight away.

However, even if you do not get as far as using pen and paper, there are very good reasons for sitting down and thinking before going straight onto the Commodore 64. For one thing, you are bound to end up with a program that has a greater chance of working both faster and is more nearly correct than a spontaneous effort. Even if for some reason it does not work straight away, you should find it much easier than usual to locate and cure the error. Obviously, the less time you spend debugging, the better. A well thought out program will perform better all round than one which was the product of an immediate attack on the Commodore 64.

Example of an unstructured program

For example, if you want to write a program to print out a specific section of memory then you might start with the idea

of inputting the start address of the area you want to print out, and then take in the length of the area. A first attempt might look something like this:

```
10 INPUT A
20 INPUT L
30 FOR I=1 TO L
40 LET X=PEEK(A+I)
50 PRINT X;
60 NEXT I
70 GOTO 10
```

Although this works, it has several serious disadvantages, but some of these may not be readily apparent until you have used the program several times.

Starting at line 10, a number is accepted, and is then put into the variable A, but the line does not inform the user what is being expected. The same problem applies to line 20. The next error is more serious. If you imagine that the user had typed in values of 30000 and 100 for the two numbers, then you will find that the numbers printed out are from 30001 to 30100, rather than from 30000 to 30099 — the hundred bytes actually required. The problem is in the scope of the FOR loop. It should be made to run from 0 to L–1, to correct the mistake. But this overlooks a simpler and more efficient solution. If the loop was made to run from A to A + L–1 then the problem of finding whether the start address was printed would not have occurred.

This will also speed up the program as it means that working out A + something has to be done only once, rather than L times. Remember that in doing this you also need to change line 40, to read PEEK I, since A has already been included in the FOR loop.

The next disadvantage is critical. In the PRINT statement no gap has been left between the numbers printed, so you will often get numbers printed over the boundary of the screen, resulting in half the number appearing on the right of the screen and the rest on the left. Again, the solution is simple: printing the numbers out say five at a time would make the output much more readable. This can be done with the inclusion of a second loop inside the first.

Example of a structured program

Other refinements to the above program would include printing the start address on the screen, to show where in memory you are printing from, printing blank lines between

each run so that separate parts of memory are kept that way, and adding a section which checks that A is within the bounds of memory. A thought out version of the program might look like this:

```
10 INPUT "START ADDRESS ";A
15 IF A<0 OR A>65535 THEN GOTO 10
20 INPUT "LENGTH ";L
25 PRINT : PRINT : PRINT A : PRINT
30 FOR I=A TO A + L-1 STEP 5
35 FOR K=0 TO 4
40 LET X=PEEK(I)
50 PRINT X;
52 NEXT K
56 PRINT
60 NEXT I
66 PRINT
70 GOTO 10
```

Note that the program could have been written like this the first time, with a little forethought, without having to go through the process between the two versions. For more on the ideas of program structure and design see Chapter 11. Meanwhile, keep it at the back of your mind when programming.

TYPING YOUR PROGRAMS

Whether you believe in structure or not, there is one thing about which you have no choice — you still have to get your programs typed in, and you have to do this through the complicated medium of the Commodore 64 keyboard. For this reason the problems of getting a program typed in correctly are examined before the more aesthetic ideas of structure and design (about which you do have a choice).

Multiple use of keys

Commodore 64 programming can give you problems from the moment you type in your first program. There are two reasons why the Commodore 64 keyboard is particularly difficult to get used to: the sheer number of different symbols on each key, and the confusion between upper and lower case letters (capitals and small letters).

In theory this multiple use of keys is practical, and it certainly makes the computer cheaper to produce, since the number of keys the manufacturer provides is less than it would have to be if each key had but a single function.

Typing the keywords

Of course things are somewhat simpler here than they might be on the Spectrum where the keywords have to be searched for and found on the keyboard before they can be used. On the Commodore 64 you can simply type in the keywords as they are needed. For most semi-competent typists it will probably be as fast typing in programs with all the BASIC keywords (in letters) as it will to use the Spectrum system.

The keyboard

The Commodore 64 has what is known as a full-travel keyboard, where each key has its own electrical connection, and is therefore independent of the others. This is obviously better mechanically than a keyboard which uses a single membrane as the electrical connection (so that the keys are not isolated from one another). One limitation of a membrane keyboard, like the Spectrum's, is that you cannot press many keys in quick succession, as some of the depressions will be lost or confused with the others. Any program suffering from this will have errors literally built into it.

Using the cursor to change lines

Even with a full-travel keyboard you should look carefully at every line you type and see if it has come up the way you wanted it to. It is much easier to change a line while it is being typed in than it will ever be to try to find the typing error when it is run. At the typing stage you can use the cursor keys to travel over the current line and change it as you wish, as well as over any other parts of the program still on the screen. Although you may feel hampered by being so methodical and slow in your typing, you should find that it allows you to get programs typed in with less errors, and therefore in less time overall.

CHECKING PROGRAMS BEFORE RUNNING THEM

This is perhaps one of the most often repeated warnings, and one of the most frequently ignored. The fact that it has become something of a cliche makes it no less relevant. It really is much easier to check a single line at the time of input than it is when it is one of hundreds and you are not even sure that this is the line to look at.

So, the moral is to look at each line as you type it in and ask yourself if it makes sense. It is actually quite hard to do

this, as it is more tempting to plug away at a program until it is all inside, and then try it out. Even worse is what happens when the program you have written causes the Commodore to crash. If this happens then you will not even have the chance to look at the program afterwards. For example, this program is a subroutine to reset small parts of memory to zero:

```
10 INPUT A
20 FOR X=A TO A + 99
30 POKE X, 0
40 NEXT X
50 GOTO 10
```

This works without problem for numbers like 40000 and 50000 which are high in memory, but if you call it with low values for A then you will lose not only the subroutine but any machine code programs, and the whole of the rest of BASIC. Your 64 will appear to be dead. In some cases you may be left with a working 64 but nonsense on the screen or in your program. Obviously, the subroutine should have included some checking, but more to the point you might have noticed the trouble if you had looked at the routine before running it.

TYPING ERRORS

There are surprisingly few ways in which you can make typing errors, but remember that it only takes a single error to destroy a whole program. Although programmers frequently talk about programs which nearly work there is really no such thing — a program either works or it does not! Unlike books, where a lot of typing errors would be unlikely to destroy the meaning, a program can be ruined by a missing full stop.

How to spot typing errors

Most typing errors, or errors in the syntax of the language, will cause the program to break down or crash. The result you see will be a message giving the line number where the Commodore 64 found that it could not continue to run. If this happens then you can LIST the line you want to see, and, with care, you should be able to see the position at which something has gone wrong, and why.

Typical typing errors

Once you have spotted the error you can edit it out using the cursor and INST/DEL keys. Most of the typing errors you are

likely to get in your programs, apart from those caused by getting letters in the wrong order, or by typing too fast, will be in the following areas:

- punctuation
- zero (0) muddled with the letter O
- line numbering problems
- editing problems
- keywords misspelt

Punctuation

At school we are all taught the importance of punctuation when writing. But most of us are not very careful with it, because we do not need to be, and in the end it may not really matter if there are mistakes. However, when it comes to computers, punctuation takes on a new importance.

Punctuation is mainly used on your Commodore 64 to let the BASIC interpreter handle some control functions. Examples of this are colons (:) which are used to tell the Commodore 64 that there is another program statement on the same line, commas and semi-colons (, and ;) which are used in PRINT statements as instructions about the format on the screen, and quotes (") which are used to denote strings as opposed to variables or numbers. Also used are the full stop (.) in decimal numbers, the dash (a minus sign –) and brackets (in expressions). It is obviously essential that the punctuation you use is correct. Punctuation marks are easier to confuse than many other characters since they are so small.

Quotes in particular can cause a lot of trouble because they must usually come in pairs, though the 64 is non-standard enough to allow you to miss off the quote at the end of a print line, closing the quotes at the end of the line for you. On most machines PRINT "HELLO will not work correctly, but on the 64 it does. However, it is always a good idea, on any line with quotation marks, to count them up and check that there is an even number.

The same principle applies to brackets, which also count as punctuation marks. With brackets you should always check that the number of closing brackets is the same as the number of opening ones.

Zero (0) and the letter O

This is a very common problem, and one that is not easy to spot. If, for example, you had a program line

```
10 F0R X=1 T0 10
```

then you would have four syntax errors in the same line. The first would be easy enough to spot because the error would appear to be in line 1 not line 10 (matters are more complex if there are 500 lines in the program), and no doubt you would also spot the use of zero in FOR and TO, and the O in 10 at the end. But because the keys are close together on the keyboard the mistakes are easy enough to make.

The problem can be compounded if you have a variable name that starts off with a zero and later appears with an O in it. Try

```
10 LET X0=1
20 PRINT XO
```

and you will get the impression that the assignment has not worked in line 10. Since O and 0 look quite similar on the screen the error can be hard to spot. Obviously, once you are aware of the danger you are less likely to make the mistake.

Line numbering problems

Whenever you write a program always try to make sure that you leave plenty of space between each line number, so that others can be inserted at a later date. If you start a program at line 100 then you have plenty of room before it to add anything that you might have forgotten (like variable declarations and function definitions — see Chapter 3). After this it is recommended that you number the program in intervals of ten or more. You will also find it easier to debug a program, when you reach that stage, if you start each new major chunk of a program at intervals of one or two hundred. This means that if you want to go to a specific part of a program then you can do so without wondering where it begins. Programs numbered 100, 110, 120 ... 200 etc, are much easier to read (and alter) than ones which start 15, 17, 18 ... 124 ....

Editing problems

The Commodore 64 has a quite useful editor, with which you can modify program lines without difficulty. It is especially helpful in allowing you to change line numbers to make a copy of a complex line which occurs in a similar form many times.

However, there is a danger inherent in modifying anything. The particular danger with editing and making copies and small modifications is that you can lose track of what you were doing in the first place. With no difficulty at all you can end up with a program that has got two FOR loops and no

NEXT statement, if, for example, the original FOR is copied into the line that contained the NEXT.

This kind of problem does not occur so much at the writing stage of a program, but while you are editing and modifying it. Since this stage can take longer in practice than the writing, it is worth being careful with whatever editing you do. The solution to the problem is a simple one. When making modifications or editing even a single line, check that each line is complete in itself, and that nothing has been missed out or added inadvertently. Secondly, check that by modifying the program you have not written over any of the existing lines, or that if you have, you have done so deliberately.

Keywords misspelt

This error is easy enough to spot, it is usually the result of typing too fast, and can arise from the mistaken belief that you are a touch typist when you are not. If you are in a hurry then you may not notice a misspelling until you run the program, particularly if the keyword printed is similar to that you wanted. This is one of the things to look out for, both while typing in a program, and also while looking at a line which has caused the program to crash.

One way around the problem is to use the keyword abbreviations as much as you can, though obviously this too can be dangerous. For example using L (SHIFT) I you get LIST, whereas using L (SHIFT) O you get LOAD. O and I are next to one another, so you do not need to be far wrong to make the error.

Another problem which can beset you with keywords is that of using them instead of variables. If you mistakenly name a variable something like LEFT$ then the Commodore will not thank you for your trouble, as this is a function name.

SAVING PROGRAMS BEFORE RUNNING

This may seem particularly obvious, but it bears repetition since it so often forgotten or ignored. There are certain problems which you can run into which will cause the Commodore 64 to crash completely (like the earlier sub-routine in this chapter, which clears memory), and if this happens you will find that none of the keys has any effect, and that to all intents and purposes the machine is dead. If this happens then there is only one thing you can do, and that is to switch the computer off and then on again. The disadvantage of this is that although it allows you to break

back into the machine (when you switch it on again), it also destroys your program.

There is no easy solution to this particular problem: since your only way of interacting with the Commodore 64 is through its keyboard, if this does not work you have no option but to switch off. In some (rare) cases all that has happened is that the Commodore 64 has got itself locked away in a long loop and cannot break out of it. If you are sure that this is the case, it may be worthwhile waiting for a time before switching the Commodore 64 off — you never know, the program may bounce back to life. In most cases, however, you are stuck with the fact that you have lost your program.

Unless, that is, you saved it to tape or disk before you ran it. It is worth doing this whenever you have made any serious changes to a program. After all, there are few things more frustrating than knowing that your program has literally disappeared, and that even if you can remember exactly what it did (and how it did it), you are still going to have to type it all back in again.

The other advantage of saving your programs (or even unfinished ideas) is that one of the laws of computing says that there will not be a power cut until you have just worked out a really good idea and got it typed in. The only time people ever switch off computers by accident is when something really important is being developed ...

Saving a program will only take a few moments of your time. With the most important programs it is worth verifying too, though if your cassette player is plugged in correctly the saving process should be reliable. Obviously, there will always be times when you press the wrong buttons on the recorder, but Commodore saving and loading is usually simpler than on other home computers.

PROGRAM LAYOUT

Returning to the theme from the beginning of the chapter, the way in which you lay out a program will make a great deal of difference, both to how it runs and to the ease with which you can debug it. Although there is not as much flexibility on the Commodore 64 as you might like when it comes to this, there are still ways of designing a program with efficiency of debugging in mind. Hopefully, when you reach the end of the book you will agree. For specific details of the ways in which you can save yourself time in the long run turn to Chapter 11.

3
BASIC RUN ERRORS

When you start programming it is all too easy to believe that everything is fine once a program runs and to feel that you are now a successful programmer. In fact it is only once your program runs that you will start coming up against the really complicated problems.

ERROR MESSAGES

When a program fails to run you will usually get an error message indicating the rough position at which the Commodore 64 found a problem: for example SYNTAX ERROR IN 25. What is more confusing is the situation in which, after you type RUN, something unexpected happens, or, worse still, nothing happens at all. The Commodore 64 appears to be twiddling its thumbs, and the familiar, reassuring READY message is no longer there.

Thankfully, there are plenty of things that you can do when you find yourself in this situation, and it is with this kind of error that most of the rest of the book is concerned. There is a series of general problems which occur with most versions of BASIC, and these are covered in this and the next chapter. Specific problem areas of Commodore 64 programming are then covered in Chapters 5 to 10.

ERROR TRAPPING

When attempting to cure a bug in a program the most important thing to do is to identify exactly where the bug is — to trap the error. In many cases this is not going to be obvious, and the problem is compounded as the bug may itself be a combination of factors or mistakes. This is particularly likely with iterative processes, where a particular portion of code is being used again and again, whether it be a formal loop, a hand constructed loop or a subroutine.

There are several ways in which you can start looking for a bug. Some of these are most useful for finding specific types, and can only be used in certain cases, while others are more general ideas and can be used to pinpoint at least an approximate area where a bug may have crept in. Some techniques to help you find a bug are given in this chapter,

while the next gives some details of what the bug might be, if it is within particular areas of the BASIC language. Other area-specific bugs are examined in subsequent chapters.

Example of a bugged program

First, however, to demonstrate the problem of error trapping, have a look at this simple program which prints the contents of memory locations 50000 to 50254, POKEs the numbers 1 to 255 into these locations, and then prints out the contents of the same locations, to check the results. See if you can spot the (deliberate) bug:

```
10 LET L=49999
20 FOR I=1 TO 255
30 PRINT PEEK (I + L),
40 NEXT I
50 FOR I=L TO 255
60 POKE (I + L),I
70 NEXT I
80 FOR L=1 TO 255
90 PRINT PEEK (I + L),
100 NEXT I
110 STOP
```

The exact location and nature of the bug will emerge shortly, but for the time being have a look at how you might set about sorting out the problem. The first thing to do would obviously be to RUN the program (unless you have already found the bug, in which case I take my hat off to you). It is likely to stop at line 60 with the message:

```
?ILLEGAL QUANTITY ERROR IN 60
```

Now you should try adding a few break points, to give you some control over which part of the program is running.

Break points

With any program that partially runs, or runs without doing what you wanted it to, one of the best things you can do is add break points to it.

A break point is any statement which stops the program in mid-flow, but which allows you to continue afterwards from where you stopped. This means that you can check that the part of the program before the break point did what you wanted it to. Once you are satisfied, you can then continue in your search for the error. In Commodore 64 BASIC the best commands for stopping the program while still allowing it to continue afterwards, are the STOP and GET statements.

The program, of course, can also be stopped by pressing the RUN/STOP key, or the RUN/STOP and RESTORE combination, but the disadvantage of this is that it leaves you to guess the point at which the program is going to be stopped, although RUN/STOP on its own does at least tell you the line at which the program was broken into. With break points you can split the program into specific sections, to see if each one is working.

GET

Using GET you can 'freeze' the program for an indefinite time while you see if everything so far has run correctly. Pressing any key will allow you to continue to the next point where you have inserted a GET. Once the error has been pinpointed to being within a small section of program you can then examine that section to see if you can find out what was wrong (see below). In the program above it would be logical to have put GETs in lines 45 and 75, after each of the first two loops. However, since the program stopped at line 60 you can be pretty sure that the error is near that line.

To use GET in these cases you must remember to put it into an infinite loop, so that the program does not continue until a key is pressed. For example, in the above program, insert:

```
45 GET A$:IF A$="" "THEN GOTO 45
```

STOP and CONT

Although the GET statement is quicker and easier to use than the STOP and CONT combination (your other option), it is also less flexible. The GET statement is very useful if you want to freeze the screen and have a look at what has been displayed so far, but does not allow you to meddle with the program, or examine it, as the first key you press will cause the program to continue on its way. After the program reaches a STOP statement you can not only examine any variables before continuing, but you can change their values too. You can also do any calculations that you need, before using CONT or GOTO line number to proceed (CONT will not always work if you have made alterations, and in this case GOTO next line number will continue the program).

RUN

If you decide that you do not want to start from scratch, but from the middle of the program, then this can be done by using RUN line number. If you are doing this, be careful not to use any variables in the program after the RUN number

that take values from the part of the program before the RUN number. When you use RUN with a line number, any simple variable information from any other part of the program already run, will be lost.

Printing variable values

Once you have found the approximate area of the program in which the error lies (around line 60, in the example above), it can often be traced more precisely by printing out the values of variables as they change. If you add extra PRINT statements to do this, you can actually see the variables as their values change, and you can get a good idea about the exact position of the bug.

In the above program, try printing out the values of the variables, using the following statement within the second loop:

```
55 PRINT "I=";I,"I + L=";I + L
```

This should show you what the error is. The loop variable I, has taken a value of 49999, and when the Commodore 64 tries to POKE it into the location I + L it crashes, because you cannot put a number bigger than 255 into one byte.

Incidentally, you can see this from the value of I + L too. This is now 49999 + 49999=99998, when you actually wanted it to be 50000. The error is actually in line 50, where the loop should run from 1 to 255 not from L to 255. The other problem you will find with this first loop is that as the second number (255) is smaller than the first one (L=49999) the loop would only be executed once, with a value of 49999.

Once you have corrected the error try running the program again. It should now run without any problem, but you will find that there is a second error. This time however it does not cause the program to crash. Can you see what the error is? Use the techniques described above to find it. (First find the area where something is going wrong, then print out the values of the variables you are using, then see if they correspond to what you were expecting).

In fact the second error is that the third loop, at line 80, should read FOR I=1 TO 255 not FOR L=1 TO 255. The problem was that I kept its former final value of 255, and L loses its old value of 49999 and now goes from 1 to 255, so the locations PEEKed are in fact 256 to 512, giving you the rather strange values you saw. You could have found the error by adding in a line 85:

```
85 PRINT "I + L=";I + L : PRINT
```

where the second PRINT simply keeps the output from this part of the program separate from the output generated as the results. The program then should have read:

```
10 LET L=49999
20 FOR I=1 TO 255
30 PRINT PEEK (I + L),
40 NEXT I
50 FOR I=1 TO 255
60 POKE (I + L),I
70 NEXT I
80 FOR I=1 TO 255
90 PRINT PEEK (I + L),
100 NEXT I
110 STOP
```

It is worth noting, however, that the use of more practical names than I for all three loops and L for the address would have made the problem much less likely to occur in the first place.

This method of finding the error by a combination of PRINTing out the values of variables and using regular GETs or STOPs is very powerful. In theory, you should know what you are expecting to happen at any point in a program, and therefore seeing the values of these variables (some of which may be purely transitory) will show you whether or not the program is running along the right lines.

When you are printing out the values of variables it is a good idea to print some message with them, as you saw in the examples above. This need only be as short as "I=". Without any indication of the meaning of what is being printed, you will soon lose touch with which variable is being used for what. It is always a good idea to print out the values of loop variables, as these are a good indication of the passage of time, and also allow you to see whether the problems are inside a loop or outside it — as you saw above.

Program listings

If you have access to a printer then the task of debugging or error trapping will be made much easier, for anything other than the very shortest programs. From listings of the program you can see its flow, and with suitable break points and variable printouts you can trace its progress. With a listing in front of you and the program running at the same time, you will be able to solve more problems faster than you can by just looking at the screen. As long as the listing is readable it does not matter how good the print quality is, and

even a secondhand Commodore thermal printer would do if this is all you are going to use it for.

Another advantage of program listings is that they can be altered by hand, so that you can see the modifications you have made to a program. This may not seem very practical, but if the modifications are not correct you will still have a copy of the original version — assuming your crossings out do not obliterate it.

Without a listing you should resort to saving every version of a long or complex program to tape or disk before making modifications, so that if you do get in a mess you can simply reload the previous version. Although this may seem obvious it is easy to waste time trying to remember what the differences were between the last three versions of the program. Remember to add a version number to the title of the program, so that you can keep track of which came first.

Listings also have the advantage of showing you just where you should GOTO or RUN from. Without knowing this you may restart from the wrong place and easily generate even more confusing errors than those you are already aware of.

Leaving space between lines

When you are writing programs it will help you later on if you leave yourself plenty of space between each numbered program line. As you could see in the program above the fact that there was plenty of space between each line enabled you to add in extra statements where they were needed. And in the example in the previous chapter that was exactly what you did.

Most of the example programs you see — both here and anywhere else — are numbered in tens, leaving nine extra line numbers between each line. You can use these if you want to insert new sections into the program, as well as those extra PRINTs, GETs and STOPs as they are needed. Without these spaces you will find it very difficult to debug programs, simply because of the numerical proximity of the statements. You may find there is literally no room to add in new lines of program, even temporarily. The problem becomes even more serious if you need to make modifications, though reliable renumbering programs are available both commercially and from magazines or computer clubs.

Extra REMs

Another feature of the Commodore 64 that can be used to great effect while debugging, is that of taking chunks of program out of use temporarily, by putting the word REM at the beginning of each statement you want to remove. You could have used this in line 60 in the program above to ensure that this was where the program was crashing.

When you come to run a program these statements will be ignored, and you therefore find out whether the error lies in the REMed part of the program or in the piece that was run. Because Commodore 64 program editing is simple you can insert and delete these extra REMs very easily, and so the process is not time consuming.

However, a word of warning is necessary. Although the idea is a simple one, it can lead to serious problems if the wrong part of a program, or too much or too little of a program, is documented out like this. For example, if a FOR loop is being edited out of the program for a short while, then it is important to ensure that both the FOR and the NEXT statement are removed. If this is not done, there will be obvious problems for the Commodore 64. Errors will be generated, or confusion caused, when it cannot find the other end of the loop.

In this example, to print out the 2 to 12 times tables you can see that the computer will actually generate another error, simply because of the extra REM which has been added at line 50:

```
10 FOR I=2 TO 12
20 PRINT I;": ",
30 FOR K=2 TO 12
40 PRINT I*K,
50 REM NEXT K
60 PRINT:PRINT
70 NEXT I
```

The REM could have been added to mask out the K loop, but since K has already taken its first value, a two times table will always be generated.

Another problem with REMs occurs if you edit out a variable assignment and then later use the variable for some reason — assuming that the assignment was made. In the above example, simply masking out line 30 as well would not have been enough, as line 40 would then assume that K was 0. If you just forget that the assignment has now been edited out, or that the assignment will affect some subsequent part

of the program, then you can go chasing errors that are not actually there, but are the result of careless or misguided editing, or documenting out sections of the program unnecessarily.

Finding logical errors

Ultimately, most of the errors in a program are going to be found by a combination of methods. Trial and error, painstaking and thorough examination of the program, methodical searching and luck are all likely to play a part. Because of this, the more you try and look at programs logically, the more likely you are to be able to find and cure the errors that are hiding there. Trial and error will play a large part not just in the finding and curing of bugs, but also in perfecting the program, once the basic idea works.

In practice, no program is ever going to be perfect. However, the more you experiment, the more likely it is to become the program you were aiming for in the first place. This trial and error and gradual iterative experimentation is particularly important in the areas of graphics, movement, and sound, where the results are to a large extent going to be subjective.

It is no longer the case that a program either works or does not. Nowadays there are aesthetic and artistic considerations which also have to be taken into account. In the same way that a painting can be said to be flawed by a tear in the canvas, a program can be said to be flawed if it contains a bug. However, there are many bug-free programs which are not pleasing to use. Experimenting at the programming stage will improve most programs, and can turn you into a professional programmer rather than someone (like most of us) who is only programming for fun.

Modular programming

The idea of structure is covered in more depth in Chapter 11, but it is worth mentioning at this stage that many of the problems you will encounter while debugging programs, could be avoided by sensible program design. Central to this idea is that of writing and testing programs as a series of small sections, or modules. Not only does this enable you to write a program which has less bugs in it to start with, but it also allows you to find the errors (which are almost inevitable) much faster, when they finally do surface. Obviously it is easier to find and cure an error in fifteen lines of program than it is in, say, fifteen hundred.

Once short sections of program have been written and checked, they can then be built together to produce the complete program. This method is used by the majority of professional games writers, as, in practice, it is almost impossible to do it any other way, once a program exceeds a certain length.

4

SPECIFIC BASIC PROBLEMS

Before going on to look at the various special areas of Commodore 64 programming (sound, graphics and movement) in detail, it is worth looking at the problems which are inherent in BASIC, the language you are using to write your programs. Each section here examines an area of the language, though some are also described in more depth in the relevant chapters ahead.

LOOPS

Loops fall into two distinct categories, and each has its own type of problems — though there are areas where they overlap. The first of these categories is the formal type (the FOR... NEXT loop), and the second is the artificial loop (where you force the Commodore 64 to repeat a section of program using decision statements and GOTOs).

FOR... NEXT loops

The FOR... NEXT loop, by its very nature, contains several obvious pitfalls (two of which you saw in the program example at the beginning of Chapter 3). However, these pitfalls will not always seem so obvious when, as a programmer, you are close to the program.

Perhaps the biggest danger with the FOR loop is that it has to contain two statements: the FOR statement and the NEXT statement. It is very easy to get these misplaced and, bearing in mind that a loop is (by definition) the section of program between the two statements, it is important to ensure that they are precisely placed. Even more problems will occur if you omit either of the two statements, or inadvertently edit out one. This means that you will literally be left with a NEXT without FOR, or a FOR loop which does not run more than once as it has no controlling mechanism (the NEXT) to send it back to the beginning of the loop.

Nested FOR loops

Another problem with the FOR loop lies in the fact that it can be nested: you can have several FOR loops running inside one another, as you did in the simple tables example above.

When setting up these, be very careful to make sure that the beginnings and endings of each loop are where you want them to be. And remember, you cannot stagger these loops as, for example, in:

```
10 FOR M=1 TO 12
20 FOR N=1 TO 11
30 . . .
70 . . .
80 NEXT M
90 . . .
100 NEXT N
```

The loops must be entirely within one another and cannot overlap. Of course the situation is confused because you do not have to specify the loop which is ending when you use the NEXT statement.

In the above case both lines 80 and 100 could have said NEXT, without a following letter. In that case the 64 would have taken line 80 to be NEXT N and line 100 to be NEXT M, making the program work correctly. It is always a good idea to have the letter in place so that you are sure which loop you are ending with the current NEXT.

When using nested FOR loops, it is also advisable to have a look at just how much work you are asking your Commodore 64 to do. If you have just four loops of 1 to 50 inside one another you are asking your 64 to go through the statements in the inner loop over six million times. Even a machine as fast as the Commodore 64 will find this a time-consuming task, and it is unlikely that you will want to put it through its paces quite that much.

Loop variables

Another area where errors can easily crop up is in the loop variable, which is the problem you saw at the start of Chapter 3. One reason why the loop variable causes problems is that most of us still use only one or two different names for the variable which controls the loop, and it is therefore all too easy to have loops inside one another that begin with the same name or use the same variables in a conflicting way.

This use of just a few different letters as the loop variables dates back to the early days of computing when Fortran was the main language used. With Fortran it was extremely difficult to use anything other than I, J, and K as the loop variables, and their use has persisted through to many

current programs. I am as guilty of this crime as most people, having been spoon-fed with Fortran while little.

Paradoxically perhaps, you will not get a problem if you have more than one loop in a program with the same variable name, as long as the loops are entirely independent of one another. Each must start after the last one has finished. Once the program demonstrating error trapping (in the last chapter) was made to work it did not matter that all three loops were called I.

Be careful also to check that the loop variable is not used as a working variable inside a loop — unless of course that is your specific intention. For example, if you have used the letter J as the loop variable it is inadvisable to try and assign anything to J, by saying LET J= something. This is likely to cause the loop to crash, and may even cause the program to stop running. Variables used as loop variables can of course be tested, examined and used in calculations. But it is important not to try and make assignments to them, and only use them on the right-hand side in an expression. It is much safer, and much better practice, to assign the loop variable to a temporary variable, and then manipulate that within the loop, rather than actually changing the loop variable in mid-loop.

The other sort of loop which can be used in your programs is the artificial loop. This is a loop you devise yourself by using a decision statement to repeat a particular portion of a program. This sort of loop is implemented formally in many languages by one of the two constructions, WHILE or REPEAT. Although neither are officially present on the Commodore 64 they can both be implemented artificially, without too much fuss.

Artificial loops

You will find yourself using artificial loops more and more as you gain experience in programming. The artificial loop is one of the best ways of keeping a program well ordered and structured — at least it is within the confines of the Commodore 64's fairly limited BASIC, a language not naturally conducive to structure.

The principle behind the artificial loop is simple. At one end (either top or bottom) there is a decision type of statement. This typically looks at a combination of factors which change within a loop. In a search loop, for example, there will be two factors, the thing you are looking for and a counter to keep a track of where you are in the array to be

searched. The search should end when one of two things happens: either the object is not present, and therefore the count will have reached the end of the possible range; or the object will have been found and the loop can be finished on this basis. The other end of the loop (bottom or top) will either be the point to which the decision branches back, or the last statement before the rest of the program, if the branch is forward.

The decision about whether the test should be at the beginning or the end of a loop is quite simple to make: if the decision is at the top then there is the possibility that the program lines in the loop may never be used, whereas if the decision is at the bottom then the program lines in the loop will be used at least once. The first of these two is the simulated WHILE loop, which repeats the loop *while* a set of conditions holds true and the second, with the test at the bottom, is the simulated REPEAT loop which *repeats* a block of code UNTIL a condition is met.

This choice of positions for tests is an easy place to make an error, and this type of error can be very difficult to find. One pointer to this being your error is when you find that the program appears to have gone round a loop in the program either one time too many (using repeat instead of while) or one time too few (using while instead of repeat).

Binary search example

A useful place to use a repeat loop is when doing a binary search. The following subroutine looks through an ordered array (A$(10)) to see if a word N$ is present:

```
100 LET I=1
110 LET J=10
120 LET K=INT((J+I)/2)
130 IF N$<=A$(K) THEN LET J=K-1
140 IF N$>=A$(K) THEN LET I=K+1
150 IF I<=J THEN GOTO 120
160 IF (I-1)<=J THEN PRINT "NOT FOUND" :STOP
170 PRINT K;" IS THE ARRAY ELEMENT WHICH
MATCHES ";N$
```

This example REPEATs the loop from lines 120 to 150 UNTIL I is greater than J. A while loop would be used in the same way, except that the test (line 150) would be at the top of the loop instead of the bottom. The best example of a repeat loop in Commodore 64 programming is the use of the GET statement:

```
20 GET A$ : IF A$=" " THEN GOTO 20
```

which says REPEAT the GET command until the result is no longer NULL. This is the simplest possible form of the repeat loop.

Artificial loops are probably more dangerous than formal loops because they have no obvious beginning and ending — just an IF statement at one end, and a pointer to the other end (within the IF statement). Also, since there is no explicit loop variable, it is advisable to be extremely careful about the variables you use. It is all too easy to use a variable twice without realising, or not at all (see below). If you have a loop like the first of the two above then you must be careful with all of the variables, I, J, and K in this instance.

Infinite loops

Another difficulty presented by the artificial loop is that it is very easy to end up in an infinite loop. Fortunately, when you have an infinite loop running on your Commodore 64 you can usually spot it, because nothing seems to be happening. In some cases, if there is an output statement within the loop then you will get quite the opposite effect: quite a lot will be happening, though it is not likely to be what you wanted and screenfuls of information with little or no meaning are common.

The cause of an infinite loop always lies in one of two places. The most common of the two is that there is something wrong with the test which should be ending the loop. Occasionally there is not even a test, which explains the error very simply. This is the case in the (in)famous program which runs along these lines:

```
10 PRINT "NIGEL IS A . . . ."
20 GOTO 10
```

Causes of an infinite loop

Assuming that the loop has a test in it then one of two things might have caused an infinite loop. The first of these is that you may have got the test itself wrong (if this is the case then it is worth reading the section on decisions and conditions below). What may have happened, however, is that the wrong variable is being altered, leaving the loop variable constant, or changing in the wrong direction. This will keep the loop going on ad infinitum, or at least until you break into it.

The cause can also be found in the GOTO statement. If these are used without caution you can end up with no

decision being taken, even if there was a decision in the original loop plan, unlike the case above, because the loop always goes back to its own beginning. The golden rule is that any GOTO statement that goes backwards in the program (at any point) should have a decision statement which can skip over it, before it is encountered again. A program which has a GOTO statement going back with no decisions between the line number before the word GOTO and the line number to which it points is bound to repeat forever, giving you the infinite loop that you are usually trying to avoid.

There are occasions when you want an infinite loop (for example if you start the program with a menu of options, and after any one return back to the menu), but they are fairly few and far between. If you do find yourself using one then be very careful indeed: therein lie many pitfalls.

ASSIGNMENTS (LET)

There are many problems with LET statements which can affect the healthy running of a program, and although a number of these may seem obvious they are worth describing anyway. It is often the obvious mistake that is the hardest to track down, simply because it is so obvious.

Variables

Although this may seem natural, whenever you make an assignment make sure that the variable you are using is the one you want to assign, and that it is also of the right type. The type is going to be one of three; a real number, an integer or a string, and it is important to make sure you are using the right one. Another problem is that if you are working in the alternative character mode with lower case letters, then upper case cannot be used for variable names. This does at least prevent there being any confusion between the two letters 'A' and 'a' for example.

Expressions

The next thing to be sure of is that whatever you are assigning to the variable name is what you want to assign to it, and that it is accurate (particularly if it is complex — a string, or a large expression). A simple mistake in an assignment, or in a complex expression, can cause problems which sometimes do not turn up until long after the program has been written.

Contrary to popular belief, once a program is running, it is not necessarily bug-free. To be sure of that, you have to

know that all separate parts of the program will work in any eventuality. You may have been lucky so far, but your luck will not necessarily hold out.

Brackets

Expressions are one of the biggest dangers, not only in assignments, but in decision making too. The danger area with expressions is that if variables and numbers are bracketed the wrong way round, or not bracketed at all, then the meaning can be ambiguous. Always be careful to make the meaning of the expression clear, with brackets. On the whole, it is better to have too many rather than too few of these.

Brackets, of course, present their own problems too. If they are not evenly matched, with the right numbers of opens and closes, then you will have more problems than you did before you started using them.

Omitting LET

One problem you can easily run into is that of leaving out the word LET from the LET statement. Although the computer allows you to do this, I would not recommend it unless you are very sure of yourself. When you are looking through a complex program some time after writing it then it is a positive advantage to have as many keywords visible as possible, as they help you to find out where any problems are and to see what is actually going on within the program.

Declaring the variables

Since assignments are the basic part of any program, it is important that they are correct. One way of helping you to be organized about assignments is to initialize the variables that you are going to use, at the beginning of the program. All you need to do is to provide a list of the variables, give them their starting values, and use a REM to say what each one is doing. Once you have this list, you will find it much easier to work out what a variable was meant to be doing, when it turns up with an unusual value. This is not strictly necessary with the Commodore 64, since any variable previously unused will be given a zero value when it is first called. However, it does make life a lot easier when you are trying to modify a program later, or find a bug in it.

USING PASCAL

If you look at any professionally-written programs, and especially those in machine code, you will find that most of

them have this list of variables at the beginning. Some languages, like Pascal, actually make you declare that you are going to use a variable, and because of this it is often easier to trace and correct Pascal errors. Naturally, your life will also be easier if you give the variables names that will trigger your memory as to what they mean. (These ideas about placing and naming of variables will be expanded in Chapter 11, when program design, and structure, are looked at in more detail).

For example, a typical program in Pascal might start like this:

```
CONST FF = 255;
      F = 16;
TYPE DAY = (MON,TUE,WED,THU,FRI,SAT,SUN);
VAR TODAY : DAY
    POS : INTEGER
    ERR : INTEGER
```

which shows how easily you can see in advance, all the variables you are going to use. In addition, you can also define your own variable types and then use variables of that type. You could assign the variable TODAY with any of the seven values given for it above (eg TODAY : = TUE).

Pascal also allows you both proper WHILE and REPEAT loops. The binary search loop above, in Pascal, would look like this (note that there are no line numbers in Pascal, and that assignments use : = instead of LET):

```
I : = 1;
J : = 10;
REPEAT K : = (J+I) DIV 2;
    IF N < = A(K) THEN J : = K-1;
    IF N > = A(K) THEN I : = K+1
UNTIL I > J;
IF I-1 > J THEN WRITELN K,"IS THE ELEMENT
WHICH MATCHES",N
    ELSE WRITELN "NOT FOUND";
```

As you can see, Pascal is very different from BASIC. However, once you get used to it, you will find it much easier and more efficient to use than BASIC if you can find a good Pascal compiler for your Commodore 64.

DECISIONS (IF . . . THEN)

The most likely places where programs go wrong are at the points where you make decisions. There is plenty of scope for error at these decision points, partly because decisions affect whether a program will go one way or another, and partly because the criteria for the decision making can be quite complex. As for other errors, you will first have to trace the bug to the right line in the program. Once you have got this far, there are only really two places where the bug is likely to be hiding, though there are one or two other places in the program which might be indirectly affecting the decision.

Acting on a true result

If the problem seems to be directly related to an IF statement, the first thing to check is the part after the word THEN — the action taken if the condition is true. For example, in the following statement:

```
120 IF X > 10 THEN PRINT "NUMBER TOO LARGE " :
GOTO 100
```

look at the PRINT "NUMBER TOO LARGE " : GOTO 100 part first.

Unless this part of the IF statement is a single PRINT message, or a simple assignment, it is likely that the program will either go to a subroutine or use a GOTO statement to redirect the program flow (the above case is very common, where there is a PRINT and a GOTO).

Redirection after an IF

In the case where the program is redirected, check that the place to which it is going is the place to which you want it to go. Did you want line 100 to be executed next, in the above example? A simple mistake in this redirection can cause an error (or errors) with no trouble at all. This is even more likely to happen if you have restructured or renumbered a program. The line numbers used at the end of IF statements are the easiest ones to forget to alter, unless you are using a sophisticated renumber routine. In essence, you simply treat everything after the word THEN as a separate program — it should work and look logical as a unit, and if it does not then that might be your problem.

There is, of course, a limit to the amount which can be effectively placed after the THEN in an IF statement. The computer only allows 80 characters to make up the current program line, and most of the first of the two screen lines

may well be taken up with the decision. You can cheat slightly by using the keyword abbreviations to get more onto the line, but be warned that it is then almost impossible to edit the line, as it overflows the 80 character limit in its expanded form. If you type in a line:

```
10?:?:?:?:?:?:?:?:?:?:?:?:?:?:?:...:?
```

up to the 80 character limit, then you can have no less than 39 PRINT statements on one line. The problem with this is that if you then LIST 10 you get six lines of PRINTs which cannot be edited. In general, it is better to use a GOSUB after the IF statement, or a GOTO to the appropriate lines of program.

Once you are sure that everything after the word THEN is correct, the only place the error can logically be is within the test (the condition or decision) — assuming that the error is actually within the IF statement. The condition is the part of the statement between the words IF and THEN.

As you know, the object of the IF statement is to allow decisions to be made in programs, by allowing one of two action sequences to be taken after a decision point. Given this, an easy mistake to make when structuring the condition is to allow only one result, rendering the decision concept redundant. Although this sounds as if it would be quite hard to do, there has already been one example described in this chapter — in the section on artificial loops. There, if the variable being tested was never changed, the result of the test would always be the same, and the loop would not end — simply because the IF statement can only go one way.

CONDITIONS

The construction of the condition or test is a place where bugs are very likely to creep in. One of the most common pitfalls is forgetting that there are not just the three standard means of making comparisons ('less than', 'equal to', and 'greater than'), but their three opposites as well ('greater than or equal to', 'not equal to' and 'less than or equal to'). The danger here is that you can end up thinking that 'less than' is the opposite of 'greater than', when in fact you must add in the vital 'equal to' case as well.

Obtaining logical results

Starting with the right criteria for comparison is also important when constructing a test. It is not difficult to find out if this is the case, as long as you know the sort of numbers

you are expecting your variables to carry. Simply work out what the result of a sample test is going to be — you could easily discover from this that the test you are using is not correct. Equally, if you think that the test you had been using is the suspect part of an erroneous IF statement, then try putting the actual values into it and examine the result.

Logical operators

The chance of an error creeping into this part of the IF statement is obviously much greater if you have two or more parts to the test. The commonest source of errors here comes in the linking together of the two. Logical operators are used to do this, and although they are simple to use in theory, the practice is quite different. Unless you think through the implications of what you are doing within each separate test, and then in the combination of tests, you can end up with results that are always true, always false, or always meaningless.

Logical operator table

There are several ways of remembering how to use the logical operators, but as long as you have the effects of AND, OR and NOT clear in your mind, you should not need to worry. Remember the following tables, if you want to work out the final result of several conditions.

In the tables 1 is *true* and 0 is *false*, and A is the result of the first test, B the second, and Y the result of the combination.

AND			OR			NOT	
A	B	Y = A AND B	A	B	Y = A OR B	A	Y = NOT A
0	0	0	0	0	0	0	1
0	1	0	0	1	1	1	0
1	0	0	1	0	1		
1	1	1	1	1	1		

The effect of this is that if you want any one of a series of tests to cause the total result of the IF statement to be true, then use OR. If you want all of the tests to be true in order to make the result true, use AND. AND is not the exact opposite of OR, but it is the logical opposite.

To demonstrate this, here are three statements which all give the same result for any value of X. They may look

different, but a close examination should reveal that they are in fact the same logical statements:

```
IF (X > 10) OR (X < 0) THEN PRINT "OUT OF
RANGE"
IF NOT(X < = 10) OR NOT (X > = 0) THEN PRINT
"OUT OF RANGE"
IF NOT((X < = 10) AND (X > = 0)) THEN PRINT "OUT
OF RANGE"
```

The test, incidentally, looks to see whether X lies between 0 and 10 inclusive. If it does not, then the message is printed.

The opposites of the three above are these:

```
IF (X < = 10) AND (X > = 0) THEN PRINT "IN
RANGE"
IF NOT(X > 10) AND NOT(X < 0) THEN PRINT "IN
RANGE"
IF NOT((X > 10) OR (X < 0)) THEN PRINT "IN
RANGE"
```

As you can see, it is not always easy to place the logical operators in their correct order and the right way round. In most cases you are going to use the first of each group of three above, since these are the simplest choices, though there are occasions when you will need to use NOT. Notice also, that although the bracketing is extensive, and in some of the cases above unnecessary, if you always put in the brackets you are less likely to make a mistake. This principle also applies to bracketing expressions in assignments, as described earlier. Often it is easier to add another pair of brackets than it is to work out the Commodore 64's order of precedence. Brackets save you any ambiguity, even if the Commodore 64 already understands. Unless you are doing your utmost to save every byte, the space saved by eliminating brackets is not worth the effort. They simply act as a safety barrier.

Before going on to have a look at some of the other problems that occur because of decision making, have another look at the six examples above. In the first case, where the decision was taken on the basis of either X being greater than ten or less than zero, look at the difference if you change the OR to an AND.

Now you are asking the Commodore 64 to print a message as long as X is *both* greater than ten *and* less than zero — obviously, this can never happen. This is the classic case of an IF statement with poor linkage between the tests,

and it is here that there is only ever one possible answer to the decision, defeating the object of the IF statement. As usual, your watchword should be caution.

MISDIRECTING THE PROGRAM AFTER AN IF

Although there are many structured languages available for micros which let you avoid many of the mistakes shown here, Commodore 64 BASIC is by its nature unstructured. To get the most from any program you have to use many GOTO statements, although most programmers will do their best to avoid them. However, when you want the Commodore 64 to run through one piece of program if a condition is true, and another if it is false, it is essential to use a GOTO in the program.

GOTO

Without the essential GOTOs in the program you can easily get the Commodore 64 doing both pieces of code in one case and one in the other. With the wrong GOTOs in place you can end up with one, neither, or both of the separate pieces of program being run, whether the IF result was true or false.

For example, the following section of a program accepts a number to be POKEd into a location, and then checks it for range, and prints a meaningful error message. If the number is out of range a new number must be input:

```
10 INPUT "NUMBER TO BE POKED POKEd (0-255)
";N
20 IF N < 0 THEN PRINT N; " TOO SMALL" :
GOTO 20
30 IF N > 255 THEN PRINT N; " TOO LARGE":
GOTO 10
```

However, as you can see, the program works well enough if N is the right size or too large, but if N is less than zero then the program will go on printing the message forever. Obviously, the GOTO 20 needs to be a GOTO 10. A wrongly placed or directed GOTO could be the problem if you find that not all of the program is being executed, or if the wrong part appears to be being executed at the wrong time.

SUMMARY OF DECISION STATEMENTS

To sum up, when looking at decision statements for a bug, there are three places to look: at the test or tests, at the action to be taken if the result of the test is true, and at the

placings and structure of the pieces of program immediately after the test or the parts of the program to which the IF statement leads. Remember that unless you specifically tell the Commodore 64 to go and run a different part of the program it will always go to the program line numbered after the one currently being looked at. It takes a branch statement of some kind to make it go somewhere else, and therefore to give you different possibilities for the outcome of any program.

BRANCHES, JUMPS AND SUBROUTINES

There is a potential danger every time you send the computer to another part of the program. This is true whether it be by a simple GOTO, a statement at the end of a decision (IF), a jump to a subroutine (GOSUB), or a formal loop (FOR ... NEXT). Since it is in doing this that programs become powerful (by using the same part of a program again and again), you are bound to use them, and are therefore bound to make mistakes.

Dangers of renumber routines

The biggest danger you will come across when you are using jumps to specific line numbers, as you do from IF, GOTO and GOSUB statements, is that if you renumber the lines in a program then you will also have to renumber the numbers used in these lines. Some of the renumber routines you can buy commercially, or those available from magazines or computer clubs, will do all this for you, but be sure that they are one hundred per cent reliable before entrusting your program to them. There is hardly anything more frustrating than making a small modification to a working program and then finding that this apparently destroys it. Even worse is when the modification is something as innocuous as a renumbering routine.

Using a trace

To find errors like wrongly numbered subroutines, or mistakenly numbered GOTO statements, it is necessary to run an imaginary trace through the program. This is most easily done if you have a listing of the program in front of you, as you can then literally run through it with your finger, seeing where all the statements lead, and where they return to.

With a listing on the screen this is more difficult, but it is still possible. This is usually what people are doing when you see them touching the screen and looking puzzled. In

practice, you will find it faster with a listing, but if you do not have a printer then obviously you will have no option but to do it on the screen.

Finding the errors in GOTO and IF statements is relatively straightforward, but the third type, GOSUB, has other problems associated with it.

GOSUB . . . RETURN

The GOSUB statement is similar to the FOR statement in that to work properly it needs another statement to complete it — the BASIC keyword RETURN. Although the subroutine without a RETURN statement on the end of it will be executed, the program will run on until it reaches the end of the program (and then report an error) or until it reaches another subroutine, with a RETURN present. Both situations will cause problems, as you can imagine. In the first instance, the program never returns to the place immediately after the GOSUB statement (the whole point of using a subroutine), and in the second at least two subroutines are used, when presumably it was only intended to run one.

This happens in the following example:

```
 50 LET IN=55795
 60 LET BA=53279
 70 INPUT "CHANGE BACK, BORD OR INK (1-3)";A
 80 IF A=1 THEN LET A$=" BACKGROUND "
 90 IF A=2 THEN LET A$=" BORDER "
100 LET N=BA + A
110 IF A < 3 THEN GOSUB 1000
120 IF A=3 THEN GOSUB 2000
130 INPUT "CONTINUE? ";Z$
140 IF Z$="Y" THEN GOTO 70
1000 PRINT "WHICH COLOUR FOR";A$;"(1-8)";
1010 INPUT B
1020 POKE N,B
2000 INPUT "WHICH SCREEN LOCATION (1-
1024)";N
2010 PRINT "WHICH COLOUR FOR";N;"(1-8)";
2020 INPUT B
2030 POKE N,B
2040 RETURN
```

Indistinct subroutines

Here you can change any of the screen location's ink colours, or the border or background colours, but because there is no line 1020 RETURN, the program is not working quite as it

should. When you use GOSUB 1000 you are using both routines, and when you use GOSUB 2000 you are using only one. In this instance it only causes confusion. When you expect one thing to happen you get two — in more serious cases it can cause errors which make the program crash. Add a line 1020 RETURN to restore normality.

Perhaps the easiest way to get an error with subroutines is to forget that the Commodore 64 will continue right into it unless you stop it by force. In the above example there was another bug, as well as the missing RETURN, in that if you let the program continue after typing anything other than 'Y', you would find that the subroutines would be run again, without a GOSUB. When the RETURN was reached the Commodore 64 would not have a place to RETURN to, so it would crash with a RETURN WITHOUT GOSUB ERROR message.

In any case where you let a program run into a subroutine without sending it there (ie accidentally instead of with GOSUB) then the program will eventually hit the RETURN (assuming this has been remembered!) and will therefore wonder where it has to return to.

Recursion

One neat trick you can use with subroutines is that of *recursion*. Recursion is when you call a subroutine from within itself. The danger is obvious. You have to include a mechanism to stop it doing this forever, otherwise the Commodore 64 will find that it has no more room to put the information onto the stack as each GOSUB is encountered.

However, this takes quite a while to happen, so do not be frightened of using recursion. To show how the simplest possible recursion works have a look at the following program which works out the factorial of a number, N (N!=N*N–1*N–2 . . . .*2*1):

```
 10 PRINT " N"," N!"
 20 LET X=1
 30 INPUT N
 40 PRINT N,
 50 GOSUB 100
 60 PRINT X
 70 GOTO 20
100 LET X=X*N
110 LET N=N–1
120 IF N > 1 THEN GOSUB 100
130 RETURN
```

Notice that you have to print N before going into the subroutine as after this its value changes. Also, you will find that although the Commodore 64 will handle a factorial up to 24 you will only get a meaningful result up to N=12 as after this the machine slips into exponential notation.

The Commodore 64 can only handle a recursive routine being called 24 times, which is why the factorial program fails to work when N has a value of 25. After 24 recursive calls the 64 will return a message saying OUT OF MEMORY ERROR. What this means is not that the 64 has used up all of its 64K, but that the stack space allotted for subroutines has run out.

Quicksort

If you can find a suitable application for recursion then it is well worth using. Although it is a very clever idea it does not have many practical applications, possibly the best being a recursive sorting routine called a *quicksort*. The advantage of a quicksort is that it is probably the fastest sort method of all — not counting those that use very fast backup storage media like hard disks to merge files together. The disadvantage of a quicksort is that the program to implement it is very hard to understand, since it not only calls itself twice internally, but also relies on a lot of extra temporary variables.

Indirect recursion

Recursion can also be implemented indirectly. If you have two subroutines, then the first can call the second which calls the first and so on. This is even more complex than straight recursion however, and by this time you really are in the realms of theoretical programming rather than the genuinely useful. There is also the problem that you can only effectively do this double recursion a dozen times before the stack space runs out.

Avoiding accidental subroutines

If you are going to use recursion at all, or any subroutines which call one another, then be sure to check that each one has its own RETURN message and that there is no scope for the program to run into the subroutines accidentally. One way around this is to put a STOP, GOTO or RETURN statement as the statement immediately before any subroutine starts (STOP in the case of program ends, RETURN if the subroutine is preceded by another, and GOTO if the program is to continue afterwards, somewhere else). If you

are having subroutine problems always check that one of these statements is present.

FUNCTIONS

The Commodore 64 uses two types of functions: those which you define yourself, and those which are predefined. And, as you might expect, each type has specific problems associated with it.

User defined functions

There are, in fact, relatively few problems with this type of function, perhaps the most obvious being that which turns up when there is a discrepancy between the variable specified in the function call and that or those used in the function definition. If, for example, you had defined a function as follows:

```
20 DEF FN(X)=Z*Z
```

then it is obviously important to note that whatever value you pass the function in the variable X, the value of the result will always be Z squared.

As long as you do not make this kind of mix-up, the use of functions should be fairly clear. In practice, user defined functions are not as useful on the Commodore 64 as they are on many machines, as they are inflexible with only one parameter passed and only one result. They can, however, come in useful if the same complex formula has to be used again and again, and they can also lend an element of tidiness to a complex or unstructured program.

Function names

Another problem which may seem obvious is that of using the same name for a function twice. This is unlikely to occur if you are using only two or three defined functions in a program, but if you are using many then you can easily get in a muddle.

Again, the answer to this problem is just to put all of the function declarations together, at the beginning of the program, to avoid confusion. You can define many functions as they can each have an identifying name of up to two letters.

Commodore 64 functions

Compared with some machines the Commodore 64 has quite a few predefined functions which you can use, but of the 26 available there are probably less than a dozen which you will use with any frequency.

Function name and keyword confusion

However, there is still the inevitable confusion between function names and the other BASIC keywords. The names are themselves keywords, and it is easy to forget that they must have parameters attached to them. More importantly, it is easy to forget that as each function produces a result it must be treated specially, either by assigning the result to a variable, or handling it in some other way (for example using the result in a PRINT statement).

Other problems with functions

When using the Commodore 64's predefined functions, it is important to remember that they produce different types of results, and accept different types of information as their parameters.

If you are having trouble using a function then it may just be because the function wants a string when you are passing it a normal variable, or that it is passing you a string and you are trying to assign this to a normal variable. Functions, like just about everything else in BASIC, allow plenty of scope for error.

Function calls may also cause the same problems with brackets that you saw earlier (with expressions and conditions), and the principle you should use is the same. Although you do not always need to use brackets, it will save you effort in the long run if you do — that way you do not even have to think about the problems of when to and when not to use them.

The hardest functions to use are the string handling ones, which are covered below. Some of the numeric functions also cause particular problems, especially the trigonometry ones which use radians instead of degrees. The integer function also causes problems as it takes the whole number less than the number passed it. This is fine for positive numbers, but negative numbers appear to get larger; for example, –12.25 is rounded down to –13.

STRING HANDLING

Any program beyond a certain level of sophistication will use strings, mainly because, like functions, they will save time in the long run. With care, you should not have too many problems with strings, but, inevitably, some will find their way into your program. One area which is particularly error-prone is that of string slicing. This is a technique to manipulate the contents of a string, and can be used both to

assign new parts to an existing string, or to create a new string of a different length.

String slicing

While slicing strings, it is a good idea to be sure of what you are up to, and to be careful when using any of the functions that are used to convert strings to numbers or vice versa — like ASC, CHR$ and VAL. However, even functions like LEN are to be used with care as what you pass it is a string but the result is a number.

String functions

Be especially careful when using the LEFT$, MID$ and RIGHT$ functions to physically slice strings. These allow you access to specific parts of a string, but remember that these are assignments, and so they cannot be used as an instruction, only as parts of one, like all the other functions on the Commodore 64. These three functions can all be usefully used with LEN to provide the end marker for the length of the current string, as well as for selecting the last few as opposed to the first few characters of a string.

Beware of using up too much space with strings on the Commodore 64. Since each letter used takes up a byte, string storage can be a costly business.

ARRAYS

String arrays

Arrays on the Commodore 64 should not generate too many errors in your programs, the only exception perhaps being string arrays, which can grow in size rapidly, with each string being anything from 0 to 255 bytes long. Even an array of length 100 will be in trouble if each of the strings grows to be 255 bytes long (over 25k!).

In the same way as maps can cause confusion, with their map references, arrays can also be worrying once they exceed the first dimension. Although the image of the two-dimensional array as a filing cabinet or chess board is helpful when you start computing, the same image causes you problems when you exceed three dimensions. It is not really necessary to retain this image or mental picture once you are used to manipulating arrays.

Multi-dimensional arrays

When using multi-dimensional arrays you can simply think of the information as being in a particular slot, which can be accessed through any one of the dimensions. In this way you

can avoid trying to conceptualize five or six dimensions. On the Commodore 64 you can actually define up to a twelve-dimensional array, though the amount in each dimension is restricted then to just two elements. However, you are unlikely to need this much because, in practice, very few of us are ever going to use more than two dimensions.

RANDOM NUMBERS (RND)

There is an argument for covering random numbers in the section on functions, as the BASIC keyword used to generate them, RND, is itself a function: RND just produces a result depending on what you pass it.

However, most of the problems with using random numbers are not caused by it coming from a function, but by the resulting numbers not being random. This may seem like a paradox, but there is no way of generating even remotely genuine random numbers without a lot of expensive hardware — substantially more than the cost of a whole Commodore 64, for even the cheapest genuine random number generator.

What the Commodore 64 does to compensate is cycle through a sequence of numbers, in what is known as a *pseudo-random* order. Most home computers resort to this method for reasons of cost and simplicity. However, you can start the sequence off in a different place by resetting the seed value. This can be done either by switching your computer off and then on again, or by calling the RND function with a positive number.

With RND(1) as well as RND(0) you can therefore give a better impression of randomness. Otherwise, whenever you run a dice program, for example, you will always get the same sequence of dice throws (using a negative value in the RND function). This could be used to your advantage in your own game, but it is fairer to throw in a RND(1) at the beginning of the game. However, it can be useful to generate the same sequence over and over if you have a bug in a program that you are trying to trace, so bear RND(–1) in mind.

Random number example

To see how RND is used in practice you can generate different cards from a pack using the following short routine:

```
10 LET Z=RND(1)
20 GOSUB 200
```

```
 30 PRINT "CARD IS ";C;" OF SUIT ";S
 40 GET A$ : IF A$="  " THEN GOTO 40
 50 GOTO 20
200 LET C=INT (1 + RND(0)*13)
210 LET S=INT (1 + RND(0)*4)
220 RETURN
```

Obviously this routine could be improved to give you the words Ace, King, Queen and Jack for C = 1, 12, 11 and 10 respectively, as well as giving the suits names, and this could then be coded as a separate short subroutine.

Random number problems

The best solution to any problems with random numbers (which are easily identified by elements of predictability creeping into the running program) is to experiment with the positioning (and therefore with the timing) of any RND(–1) and RND(0) statements. Rearranging them, or even shifting them by one or two statements, can create the illusion of randomness that you are looking for, since they will not then be at the same position as before.

One of the most common areas of difficulty with random numbers is in handling the result. It is easy to forget that random numbers are generated between 0 and 1, but although they can be 0 they can never actually be 1. This means that if you take INT (RND(0)) you will always end up with 0 as a result. To get a result of either 0 or 1 you will have to take INT (RND(0)*2) instead — in the example above, 1 was added to the results so that although the random number fell between 0 and 12 in one case and 0 and 3 in the other, the correct results were those actually printed (1–13 and 1–4).

Obviously, there are many other areas of BASIC which will cause problems, but these are dealt with in specific chapters from here onwards, under the areas of programming to which they are relevant.

5

INPUT

Commodore 64 chips

At the heart of the Commodore 64 there are several very important chips. There is the 6510 main processor (a variant on the 6502, with a similar instruction set), the 6569 VIC II graphics controller (video interface chip), the 6581 SID which handles the sound (sound interface device) and assorted RAM and ROM chips. These are the hardware devices which do most of the hard work and handle much of the information that goes round inside the 64. However, although the 6510 does some cassette driving, none of these chips is dedicated to input and output — interfacing. This task is left to a pair of complex and powerful chips called the 6526 CIAs (complex interface adaptors).

ABOUT INPUT AND OUTPUT

Input and output are crucial functions of your Commodore 64, since it would be quite useless without them. Input is the process which allows you to communicate with the Commodore 64, via the keyboard, cassette recorder, disk drive, joystick or modem — or any other device, through the user port. Output is equally important since it allows the 64 to communicate with the outside world: this includes the screen, printers, modems, disk drives, cassette recorders or any other peripheral, via the user port.

Much of the Commodore 64's input and output (I/O) is done automatically, saving you a lot of time and effort. For example, you do not have to keep checking the keyboard to see if a key has been pressed, or even work out what the key was from its position on the keyboard. But if you want to take control of these functions then there is sufficient documentation in the *Advanced User Guide* (or certain of the better books on the Commodore 64) to allow you to do so. The advantage of taking control of parts of the input and output is that you can then treat it in the way you want rather than that automatically used. For example, you may want to redirect output from the screen to a printer instead.

Should you really want to, you can take control of all the input or output to any device, simply by knowing the

commands to use (from BASIC or machine code) and the addresses in memory which handle the incoming and outgoing information. In most cases you will probably be quite content to let the 64 do as much as it can to help you, only taking control when you want to change the way the system behaves.

Input and memory

All information coming into your computer has to be stored in memory (if it was not stored it would simply disappear, rendering the exercise of sending information to your computer useless). As a result the handling of this incoming information ultimately consists of knowing where in memory the incoming information is going to be stored. This, of course, is at the lowest (and therefore the most complex) level, and is what you would have to do through machine code. Most of us need never go further than using simple commands from BASIC. However, even these BASIC commands which assist you with input are riddled with danger.

The main peripherals you are likely to use for input are covered first. The BASIC commands which accept input from all peripherals in more complex applications are described next. These include the file commands CMD, OPEN and CLOSE, as well as the physical input statements GET# and INPUT#.

CASSETTE RECORDERS

The cassette is probably one of the very first things that you used with your Commodore 64, since it is the easiest way of getting that vital first program into the machine. For most people this will have been a games cassette — unless you were lucky enough to have a disk drive from the start! Soon after plugging the Commodore 64 and cassette recorder together you probably found that you can occasionally get loading problems. However, the Commodore cassette unit is among the most reliable around. This, coupled with the fact that programs are saved twice consecutively (automatically, and one of the reasons why they seem to take such a long time to load) means that loading problems should be few and far between.

Loading programs

The difficulty of loading programs from cassette depends on the quality of three things: the recorder, the cassette and the recording. If the cassette is commercially recorded then there should be few problems unless your recorder has got

very dirty (regular cleaning and demagnetising can work wonders). You may encounter problems if a cassette has been made on one recorder and is being played back on another, though most Commodore units are almost identical. If you are using a non-Commodore recorder with an interface then you will get the best reliability by keeping one recorder specifically for the computer. This way you do not have to worry about resetting volume or tone controls every time you use it, or, alternatively, forgetting to reset them and getting poor quality or unreliable recordings as a result.

Loading in BASIC

The one BASIC command used for input with the cassette unit is the LOAD statement. LOAD is simple to use and the only thing that you have to watch out for is the type of file you are loading in. If it is a standard PRG file then there is likely to be no problem, as long as you remember to use LOAD "name", 1, 1 if you are loading machine code back into the place from which it was saved.

Loading in machine code

If you are loading in machine code that is not from a commercial game then it is a good idea if you know something about it before you load it in. Since the machine code appears to the Commodore 64 as a block of data, and could easily seem meaningless, it can help a lot if you know the position to which it is being loaded, and the extent of the code.

Knowing the size and position of the code you are loading in makes it easier to spot mistakes. If you know that you have some machine code then it is very unlikely to take up 20 or 30K, since most machine code routines fit quite comfortably into several hundred bytes. However, if you are loading in stored screens or other large graphics then you may need to load in large chunks of memory. If possible try to find out something about the code before you load it in.

The only other problem you are likely to find with LOAD is that if you have a lot of programs on tape then you can have trouble locating the right one. If you know the program name then you can easily find it by loading in the name, but if you do not you may just have to go through the tape program by program, loading each one in and looking at it to see if it is the one you want.

Keeping a catalogue

Obviously your task is going to be much easier if you know what the programs are called and what they do. To this end you can save a lot of effort by keeping a list of the programs on a tape on the inlay card. It does not take long to do this, and the effort will save you a lot of time and trouble later on. This may seem obvious, but it is all too easy to develop a program and save versions without keeping some kind of record of what they are.

Even if you have a tape with no catalogue, it is easy enough to make one. Try to load in a program name which you are sure does not exist — like '**zzz**', for example. The Commodore 64 will then go through the tape and each time it finds a program will put the name up on the screen for you. Next time you want to load something off the tape you need only look through the list you have made of its contents.

Using REM

An even better aid than a catalogue is making the first line of any program a REM statement saying what the program does, along with a date. This way you can differentiate between different versions of the same program. The date can also be encoded into the program name on the tape. This gives you a reliable guide to the version of the program you are using.

DISK DRIVES

Disk drives act as input devices in almost exactly the same way as cassette recorders, though of course they are much faster. You are most likely to have the standard 1541 drive, which is the easiest to use, though you may be lucky enough to have an interface and drives from one of the Commodore business machines. In any case the load command is the same as for the cassette unit, except that you must specify that you are using disk by adding **,8** to all of your commands. For example:

```
LOAD "Filename",8
```

SEQ

In addition to the simple LOAD command you can use other commands with both the cassette unit and the disk drive. These enable you to have files on tape rather like arrays, and these can be read or saved one piece of information at a time from SEQ as opposed to PRG files on tape or disk. The commands which allow you to do this are covered in a

separate section, towards the end of this chapter. The use of these commands is straightforward, and not much different from inputting from the keyboard (see below), as long as you are confident in the use of the various channels and device numbers (see below, or the various user manuals for more details of these).

Handling disk drives with care

When using Commodore disk drives (especially the 1541) it pays to be very careful with them. They are probably going to be the most fragile part of any computer system you have, and can easily become misaligned. If this should happen, or if you find that data is getting corrupted then do not on any account keep trying to record. Get the drive repaired first, because otherwise you are in danger of destroying anything else you may have already saved onto a disk. You can get problems if you keep switching the computer or the drive on and off with a disk in the drive, and even a knocked mains plug has been known to destroy the data on a disk. Treat them with caution, but do not let this put you off getting one. The improvement in speed over cassettes makes the care with which they need to be treated well worth the effort.

Importance of backup copies

Although it seems obvious, remember to take regular backup copies of anything you want to keep. It is even worth taking tape copies of the most important files, just in case the disk drive should go wrong, or the information gets lost from the disk. Remember also that the disks will eventually wear out, and it is therefore risky to use a very old disk — far better to invest in a new one.

Specific disk commands (DOS)

Disks also allow you a whole range of specific commands via the disk operating system (DOS), and it pays to be as careful with these as you are with the disks themselves. In particular, any commands which allow the possibility of destroying information on a disk, like those for formatting or deleting, should be treated with special care. Before using any commands of this type it is always a good idea to have a look at the disk directory, so that you can see whether the files to be destroyed or the disk to be formatted actually contain any useful information which you might like to keep. If you are uncertain about a file then have a look at it by loading it in. You are sure to find out that it *was* important if you just go ahead and erase it.

KEYBOARD INPUT

The keyboard is probably the most important of all the input devices, simply because without it you cannot interact with the Commodore 64 — unless of course you already have a highly sophisticated joystick input routine, which allows you to dispense with the keyboard altogether.

Generally speaking, all other input devices are controlled by what you type in on the keyboard — even if this is only a program which allows another peripheral to take control, dangerous though this is.

The way in which the Commodore 64 handles the information sent by the keyboard will be described in a moment, but first have a look at the BASIC commands which allow a program to collect information from a user, through the keyboard. The two commands used are GET and INPUT, though the PEEK command can be used if you want to be very clever and you know how the keyboard input routines work internally.

INPUT

You should not have too many problems getting information into the Commodore 64 from the keyboard during a program run. However, one common source of error can be found in the INPUT string. If you want a message printed then you can put one in front of the name of the variable to be input, but remember that this cannot be longer than 38 characters.

Logical screen line

All input must take place within the current logical screen line which effectively limits all string input to 79 characters, even if a message is not printed with the INPUT statement. If you do accidentally go over the current logical screen line then you will find that the only input accepted is that typed after the current logical screen line. This can look confusing if you do not realise what is happening.

When using the INPUT statement beware of an unusual feature of the Commodore 64 which is that if you do not put in any value for the INPUT variable(s), and simply type RETURN then the previous value(s) are taken, rather than nothing. Once you are used to this feature then you will probably find it to your advantage, but at first it can be confusing.

Unlike many computers, the Commodore 64 will not crash if you type in a string when it is expecting a number. Instead you will get a REDO FROM START error, and the

INPUT line will be returned to its original value, ready for you to type in fresh answers.

Breaking out of programs during INPUT

Another problem you can find with INPUT is that you cannot break out of the program simply by using the RUN/STOP key. To break out you have to press the RESTORE key at the same time, which has the disadvantage of clearing the variables, and resetting several pointers, though fortunately it does leave your BASIC program intact.

INPUT has other disadvantages too. It cannot be used in direct mode programming, unlike most other BASIC statements, as it uses the same buffer, and you will therefore get an ILLEGAL DIRECT ERROR if you try to use the two together. In addition, you cannot input either a comma (,) or a colon (:) directly, though if you want to use these you can input a literal string by putting a quote mark before the string to be input. Note that if you do this then quotes in the middle of the string will cause the 64 to crash with a FILE DATA ERROR.

For the reasons mentioned above, you will not find INPUT widely used in commercial programs: GET and PRINT are used instead. There is a lot to be said for validating the characters which come in one at a time, though it does require a lot more programming effort. Certainly for everyday use and for most applications INPUT is very useful. It is only when a messy message on the screen is unacceptable that other means are best employed.

Inputting numbers as strings

Finally, one hint which can save you trouble is to input all numbers as strings, and then use VAL to convert them into numbers. This will save you the problem with REDO FROM START mentioned above:

```
INPUT "AGE";A$ LET A=VAL(A$)
```

GET (keyboard buffer)

GET, by comparison, is relatively simple. The only thing to remember about it is that it does not wait, like INPUT, for you to type something, but looks at the keyboard buffer straight away, and returns the first result from it. If the buffer is empty then a null string is returned. The practicalities of this are that if you want to look for a single key press then you can do so by having a loop (a WHILE loop, in effect) which looks at the keyboard until a key press is put into the keyboard buffer:

```
10 GET A$ : IF A$=" " THEN GOTO 10
```

This will jump out of the loop as soon as a key is pressed. However, if there was already a key press stored in the keyboard buffer then this would be returned rather than the new key press.

You can get round this problem by clearing out the keyboard buffer first. This can be easily effected with a second loop, inserted before the first one:

```
5 GET A$ : IF A$<>" " THEN GOTO 5
```

This waits while no key is pressed, and then the second loop comes into effect to detect when a key is pressed. Often you will only need the loop in line 10, but to avoid confusion it is best to clear the buffer before GETting a key press. You can easily find that a key has been pressed earlier twice by mistake, or even that something has fallen onto the keyboard accidentally. If you want to get specific single character input then you should use something based on the following routine:

```
10 GET A$ : IF A$<>" " THEN GOTO 10
20 PRINT "WHICH PROGRAM TO RUN (1 OR 2)?"
30 GET A$ : IF A$<>"1" AND A$<>"2" THEN GOTO 30
```

This will continue to accept input until either 1 or 2 is pressed, and is best coded as a subroutine so that you can call it from anywhere in your program.

JOYSTICKS

Sooner or later you are likely to use your computer with joysticks, and this really presents you with very few problems, since with commercial games they either work or they do not. However, if you want to write programs yourself which use joysticks then you must first understand a little more about the hardware of the 64.

CIA chips

Most of the interfacing is controlled by two CIA chips. The first of these (CIA 1, not surprisingly) deals with the keyboard scanning and also with the input from the two joystick ports. When joystick port number two is being used then the keyboard is still scanned by CIA 1 as usual, every 1/60th of a second, but when joystick port number one is used then CIA 1 cannot tell the difference between the closing of the fire button on joystick one and the pressing of a key on the keyboard. One effect of this which might cause confu-

sion, is that after using joystick port one you may find the keyboard buffer is full of rubbish characters. If you use GET, you can see what these characters are.

Joystick ports

The result of this is that if you want to use one joystick and the keyboard at the same time then you should program the joystick for port number two (or B). If you want to use two joysticks then you may have trouble reading the keyboard. If you are using a light pen (and provisions have been made for this in the VIC II chip) then it must be plugged into joystick port one (or A), effectively cutting out any serious use of the keyboard.

Joystick direction bits

When you are reading the joystick ports (effective memory addresses 56320 and 56321) remember that it is the bits you want to look at and not the bytes, so you will need to mask out the bits that you do not want.

If you use 15–((PEEK 56321) AND 15) then you will get the reading from the joystick direction bits, and these are equivalent to the following directions:

15–((PEEK 56321) AND 15)	Direction
1	North
2	South
3	—
4	West
5	North West
6	South West
7	—
8	East
9	North East
10	South East

Joysticks and machine code

It is unlikely that you will have any other serious problems using joysticks, though it is worth mentioning that it is all too easy to go off the borders of the screen, and therefore make an error when plotting the joystick position. If you are writing any kind of serious joystick application then you may well find that BASIC is too slow. If you find this to be the case then you should get hold of a good machine code driver routine for the joysticks. One is available from the Commodore *Programmer's Reference Guide*, though there are also many other sources.

USING SEQ FILES

Before going on to look at the problems you encounter when getting information into your computer from other peripherals, it is worth making a few notes about using SEQ files on tape and disk. SEQ files allow you to store information away in small chunks, and to retrieve it in the same way. This has obvious applications such as keeping a mailing list or even an index of your record collection, and is a very powerful feature of your 64.

To use SEQ files you must first master the general file handling commands which are OPEN, CLOSE and CMD. You can then use the INPUT#, or GET# commands to input information from previously created files. Details of using PRINT# to send information to these files are included in the next chapter, along with details of all output from your computer.

CMD

CMD, as you will probably have found out, is an unusual command, in that it looks more like a patch into the system than an integral part of the Commodore 64's capabilities. It can give unpredictable results in that it does not always work — particularly if you use GET between two PRINTs which have been redirected. You can get away without using it simply by OPENing a logical file number and using PRINT# instead of PRINT to output information. This has the added advantage of allowing you to use the screen as well as the output file. It is best to restrict the use of CMD to dumping listings to printers and the like, rather than to redirect output from the screen to a logical file.

OPEN and CLOSE

OPEN and CLOSE are both easy commands to use and understand, and are therefore generally problem-free. One problem which you may encounter, however, is that of forgetting to CLOSE a previously OPENed file before a program finishes. The result of this will be that the last block of data will not be sent to the output file, and you are likely to have difficulties the next time you try to read the file. It is always good practice to include the relevant CLOSE statements at the end of any program which uses file handling.

Logical file numbers

Another easy trap to fall into is that of forgetting that although you can generally open any logical file number from 0–255,

the file numbers from 128–255 will automatically have a carriage return and line feed sent to the output file at the end of every line. This is not usually desirable, and in the normal course of events 0–127 should be used. Although you can use any of these numbers, remember that you cannot have more than ten logical file numbers open at a time, and that you cannot have more than three channels open to a disk drive.

Command channel (opening and closing)

Note also (when using logical files and disk drives) that when you close the command channel (15) you will close all other files which are opened to the disk drive. The moral of the story is that you should open the command channel at the beginning of a program which is to use it, and then close it as near to the end of the program as you can — at least after the last disk access.

GET and INPUT

Once you know how to use the standard forms of GET and INPUT then you should have few problems with the complex forms GET# and INPUT#. As long as you remember that the logical file numbers you use must have been opened first, and that you are effectively moving pointers along inside a file rather than depending on keyboard input then you should find them easier, if anything, to use.

Remember that now you do not have to use a loop around GET# as it is only the next character in the file which will be picked up. Remember also that INPUT# no longer needs (or can use) an input string, and that the input no longer depends on the logical screen lines, but rather on the punctuation within the file which is being read from. A semicolon(;), colon(:) or a comma(,) will stop the current variable being input from the file, as will the character for carriage return (13). Even though you are no longer restricted by the logical screen line you still have a maximum length for strings of 80 characters including the terminator.

OTHER PERIPHERALS

Once you know how to use the logical files and the commands associated with them you will have no problem applying this knowledge to any other peripheral which you want to attach to your computer. As a result of this you should find the majority of available peripherals very easy to use. Another reason for this is that a peripheral is far more appealing to the buyer (you) if you can just plug it in and use

it, without having to resort to lots of complex programming in order to be able to use your new piece of equipment.

Most of the Commodore peripherals require little effort to be made on your part: the printers will print using the Commodore 64 graphics characters, the disk drives and cassette units will keep your information for you in a simple and efficient way and so on. However, remember that any peripheral not actually designed to run with the Commodore 64 (and many printers, modems and the like, fall into this category) may not be entirely compatible.

Compatibility

This is really something you ought to find out before buying, as there is often nothing you can do to solve the problem. On occasion you will be able to write a machine code routine to interpret the messages which come into the Commodore 64, and to send out the correct replies, but, as you can imagine, this is not a simple solution. In fact, using this method you can set up any serial and parallel communications you like to any peripheral using the second CIA 6526 chip, as long as you know the protocols you want to use. Both Centronics and RS-232 are feasible, but note that the Commodore RS-232 runs at a different voltage to the accepted standard (5V instead of the usual 12V) and you can therefore damage your computer by attaching peripherals without extreme caution.

The sort of peripherals which interface into the Commodore 64 through the user port are likely to be easy to use, and in any case should come with detailed instructions about how best to attach and use them.

A typical example of this is a Centronics printer interface, which should come with its own machine code programs to handle information to and from the printer, as well as allowing you to designate special effects yourself like double line spacing or enlarged output, if the printer is capable of it.

Peripherals like an interface should certainly come with programs which help you to avoid doing the input and output yourself. However, as was mentioned earlier, enthusiasts could do any input and output through the user or serial port themselves, using the second CIA 6526 chip.

Other peripherals which will be pre-programmed to cope with the Commodore 64, are speech units and modems, the latter of which are getting more popular all the time. Specific problems with a non-Commodore peripheral should be referred to the dealer or manufacturer from which

you bought it as they should be in the best position to help you.

PEEK

Although it is often necessary to resort to machine code if you want to handle specific input yourself, the one BASIC command which is essential is PEEK. If you know the addresses where information is stored when it arrives then you can use PEEK to examine the locations, and with the controlling locations (like 56321 for the joystick) you can mask out parts of the resulting byte to show you the precise bits you want to examine, using the logical functions AND and OR.

PEEK can be used to monitor what is coming in through the keyboard, for example, by using it on the keyboard controlling locations. This program prints out the keyboard matrix number of a key pressed:

```
100 PRINT "TYPE A KEY"
110 GET A$ : IF A$="  " THEN GOTO 110
120 PRINT "MATRIX NUMBER OF ";A$;"
IS";PEEK(197)
130 PRINT "SHIFT FLAG FOR ";A$;" IS";PEEK(653)
140 PRINT "RVS FLAG FOR ";A$;" IS";PEEK(199)
150 PRINT "ASCII CODE OF ";A$;" IS";ASC(A$)
160 GOTO 100
```

SYSTEM VARIABLES

The system variables are locations in memory which the Commodore 64's operating system uses to keep track of certain internal variables. There is a list of these and their functions in the *Programmer's Reference Guide*, and they can be PEEKed or POKEed just like any other memory location.

From an input point of view, things that are going to be relevant are the values for the length of the keyboard buffer and the keyboard repeat function. Note that if you do decide to tamper with the system variables then, before you do so, make sure you save the program as it is. Making a mistake while changing the system variables is probably the easiest way to crash the Commodore 64. And it is not always easy to work out why this has happened. Even if you do something as simple as making the keyboard buffer longer than its standard 10 characters (POKE 649, new length), you can cause confusion between the keyboard and screen handling routines.

READ, DATA and RESTORE

Input need not necessarily come from an external source, and there are three BASIC keywords used to input data from within the program: READ, DATA and RESTORE. These are some of the simplest words which can be used in the BASIC language, and you should not find many problems. Even when problems do occur they are usually fairly easy to solve.

Perhaps the commonest problem is that of having a discrepancy between the amount of data read in and the amount present in the data statements. It is easy enough to check that you read the same amount as you are meant to, but you can make this even simpler if you have the same number of pieces of data in each data statement. This way the number of pieces of data that you have altogether can be quickly calculated.

Finding the number of pieces of data read by a program is more difficult. In the first place, you should check to see if the word RESTORE has been used. If it has then start your count after it. However, you can even run into trouble using this method, as there may be too much data read in even before the RESTORE command. Next you should look for the READ statement(s). These are usually used within loops to make the reading process easier, so you should multiply the READ statement by the number of times the loop is used. This is all very time consuming, and can be avoided if you check at the time of writing that your READ and DATA statements match up.

The Commodore 64 can help you with DATA and READ problems if you look at the system variables stored at locations 63 and 64. These will give you the program line number which the last DATA statement was read from. Simply multiply the two together thus:

```
100 LET DL=PEEK(63)+256*PEEK(64)
```

and look at DL to see what the DATA line number was. This can be useful not only in tracing mismatches between the number of READ and DATA items, but also in correcting large amounts of data like sprites or sound programs (both of which are covered elsewhere in this book).

6

OUTPUT

ABOUT OUTPUT

Output is as vital to the Commodore 64 as input. Without output you would get no feedback from your programs — no results. More to the point, it is very unlikely that you would ever have been able to type in a program in the first place — without the aid of a screen. Output, however, although very simple in theory (just a question of sending the bytes you want to certain memory locations), is complex in practice.

Fortunately, most of the work is done for you by a combination of the operating system and the two CIA chips (see the previous chapter for details of these). Information which is due to leave your computer obviously needs quite a lot of organizing before it can be successfully sent. The bytes must be doctored: in some cases extra bytes are added to act as controls and safety precautions, and in others control characters (such as line feeds) have to be embedded where they are needed. This applies to output to all of the various output ports, including the TV modulator (or monitor), the cassette port, disk drives and the user port.

As with input, it is possible for you to do this kind of work yourself, but most of us are quite content to leave it to the Commodore 64. However, even if you leave the hardest parts alone, there is still a fair amount of effort required by you in outputting information from BASIC.

CASSETTE RECORDERS

The last chapter looked at the problems of loading information into the Commodore 64. In fact the problems with the output commands are similar, in that you are still communicating with an outside device, and the aim is the same but opposite — the safe transfer of information out of the Commodore 64 instead of into it. Naturally, if you know that a program is safely on tape (and you can find this out by using VERIFY) then you are going to be more likely to be able to load it back in again at a later date.

The way you save programs or data onto cassette is by using the SAVE command, as you will have discovered within a few hours of switching on your Commodore 64 for

the first time. This command works well, as long as you remember certain things about it.

SAVE

Firstly, you should not forget that you must always give a filename to the file you are saving. Although LOAD will work without one, the SAVE command needs to have some identification for the file, so that it can write a header identification file onto the tape or disk. In practice, you can call every file the same name (doing this on disk is inadvisable — for obvious reasons!), but you will only end up confusing yourself, as, when you come to a tape with fifteen programs all called PROG, you are not likely to remember which was the one you actually wanted to load back in.

Logical filenames

This may all seem natural, but it is easy (out of laziness) to end up with meaningless filenames, where meaningful names will save you time and effort. The principle can be extended by calling your BASIC programs one sort of name, and calling machine code files (saved as SEQ rather than PRG files) something different. It is a good idea to establish a protocol in this way and then keep to it. All of my machine code programs start **mc...**, while blocks of RAM start with **c...**. If any file comes up that does not start with one of these two prefixes then I know that it is going to be a BASIC program.

Saving machine code is rather more complicated than saving BASIC programs. You will be using a machine code editor/assembler or monitor if you are doing this, and to be accurate it is best to follow the instructions for your particular software. Saving blocks of memory can only be effectively achieved from machine code, and if you want to do this then it is recommended that you look at the wide range of machine code software tools available.

VERIFY

Once a block of RAM is saved onto tape or disk then VERIFY can be used to match it with whatever you have in memory by adding a 1 to the statement:

```
VERIFY "NAME", 1, 1
```

or, for disk drives,

```
VERIFY "NAME", 8, 1
```

This would compare what was on tape or disk, byte by byte, with what was currently at the position in memory specified

in the header block. Without the ending, 1 the statements above will simply compare the named program on tape or disk with the current BASIC program in memory. Note that as VERIFY is a byte for byte comparison the program in memory must be identical to that on tape or disk: one byte more or less will cause the familiar VERIFY ERROR message to appear.

Problems with saving

It has probably been said a thousand times, but it still bears repetition, if only because the same mistake is made again and again: *verify everything you save* before accepting that it has correctly gone onto tape. It is all too easy to make the assumption that if you have typed the SAVE command and seen the screen go blank for a while then the program is safely on tape.

There are many reasons why it may not have been safely saved. For one thing, even if you type SAVE and the tape recorder is not attached you will still get the impression that something is going out of the Commodore 64. In fact it is, it is just not being received by the tape. Other problems can occur if you are using an interface and a non-Commodore cassette recorder: the record level may be too low or too high (giving distortion), or you could even have the leads in the wrong way around.

Moreover, when using any recorder other than Commodore's own, both saving and loading are generally more effective if you only have one of the two wires plugged in at a time, so that you are either using the load wire or the save wire but not both. Although this means that it is easier to forget to switch the wires back and forth between loading and saving, it does give you better recordings, and a better chance of a successful playback. The reason for this is that there is interference between the two wires which can add noise (feedback) to a recording. This problem is avoided on Commodore's own unit because it has heavy shielding on any wires leading to the cassette unit, thus preventing interference and feedback.

This is all probably below your level of competence, but since so many programmers (whether at home or professional) make the same mistakes, it is worth repeating. Do not ever assume something is safe without checking it. Even something as simple as pressing play (instead of record and play) on the cassette recorder will destroy any chances you have of making a safe copy of what is in memory.

DISK DRIVES

Disk drives can be treated similarly to cassette recorders when it comes to output, even to the extent of using the same commands (with the ,8 added to identify the fact that disk rather than tape is being used). Because disk drives are more sophisticated than cassettes do not be tempted into skipping the verify stage, or into making the assumption that the program will save correctly every time. Disk drives are prone to errors too, and although they are usually less of your making than those you get with cassettes, it is always worth checking that the file is what you wanted and has saved to disk correctly.

If the heads on the disk drive have shifted even slightly out of alignment then you are almost certain to get loading and saving problems. If you find this happening do not under any circumstances start trying other disks in the drive. If the drive is capable of destroying information on one disk then it is quite capable of destroying information on all of them. Instead, take your disk drive along to a dealer, who will be able to realign the heads for you without too much trouble. If you are confident in electronics you can perform this task yourself, but this is not recommended unless you know exactly what you are doing.

Logical file handling commands

As well as the BASIC commands which you can already use with cassettes there are other BASIC commands built into your semi-intelligent drive, and these give you extra control over disks which you do not have over tapes.

These commands are accessed through channel 15 which gives you direct access to the disk operating system (DOS). Beware of using commands like:

```
OPEN15, 8, 15"N0:TITLE, DI":CLOSE15
```

which formats a disk, creating a new block availability map (BAM), directory and timing markers. The effects of not being careful can be truly disastrous. If you use the above reformat command on a disk which contained valuable programs you will not be pleased! And because disk drives are so much faster, the erasure happens very quickly too.

Importance of backup copies

Until you have lost a really important piece of work you will probably not take much notice of all this talk of saving meticulously, checking it has saved correctly and using the

deletion and formatting commands with care. Once you do lose that vital piece of work you may, like me, come to the conclusion that there are benefits to be had from saving all your really important programs at least twice.

A simple way of doing this with cassettes is to record the program at the beginning of each side of the tape, which means you do not have to use too many tapes. The advantage of doing this over recording the two copies one after the other is that occasionally a tape gets eaten by a tape recorder. When this happens you often get a break in the tape and the first portion of the tape may easily be destroyed.

However, if you take the cassette apart, throw away the bit of tape that has been chewed up and reassemble the case. You can turn it over and rewind it, and the program will still be there. Your second copy of the file would be pretty unlikely to survive a tape chewing exercise if it was right next to the first copy on the tape.

SEQ and REL files

When you start using disk drives for serious applications you will probably start using sequential (SEQ) and relative (REL) files, the two types of file which allow you to store pieces of information rather than whole programs. SEQ files can, of course, also be used with cassette if you want to write records of information which can be read back later, but using disk you will find the true worth of such files. Using tape you will find the computer is really held up by the relatively slow speed of the tape; with disk the delay is substantially less.

These files use standard BASIC commands like INPUT#, GET#, OPEN, CLOSE and CMD for input and channel handling, and PRINT#, for output to the files. When using either a cassette unit or a disk drive you must assign it as a logical device, after which all PRINT# statements addressed to that channel will cause output to go to the specified file.

Simple PRINT statements (as opposed to PRINT#) can be redirected to a channel using the CMD statement, but this is not entirely reliable so using OPEN and PRINT# is recommended as a safer solution. CMD is effectively used to send listings to a printer or disk drive, but on the whole should be avoided.

With SEQ and REL files you should take special care to ensure that they are opened and closed correctly. The

OPEN and CLOSE commands do this for you, and as long as every file opened is later closed then you should have no problems. If you forget to close a file you can be left with files on the disk which are locked open. Once in this state the files are very hard to handle successfully, and are often irrecoverable.

Separators with SEQ and REL

When using files like this it is also a good thing to remember that strings should be made to end with a separator or a carriage return character. You can do this by adding CHR$(13) or commas to strings going out via the redirected PRINT# statement. You can also add a line feed character automatically by using channels from 128 to 255 rather than the more common numbers up to 127. Without separators between different parts of information, files can just become a jumble of meaningless unseparated letters and numbers. On disk it is more efficient to use relative (REL) files with which you can define the lengths of the individual fields within records. With REL files you can achieve more serious applications like address files, giving more power to your computer.

SCREEN OUTPUT

The most important area of output is not the peripherals (though of course it is important to be able to store your programs) but the screen. Most of the screen output is actually done for you, by the operating system. For example when editing a program, or typing one in, the operating system makes sure that what you type is displayed. Some of this is simple, but other parts, like the changing of colours or the expansion of abbreviated keywords to their full forms are more complicated.

Fortunately, you do not need to worry about most of this, and as a user you only need to know a small number of output commands, PRINT and POKE being the two most important.

PRINT

PRINT is another of the many Commodore 64 commands which seems simple on the surface but which can involve many problems. One of the particular difficulties with PRINT is that it (almost) has arguments, or parameters. You have to decide what to feed the command (ie what you want to appear on the screen, and where you want it to appear). There are controlling punctuation marks and functions which

you can put into PRINT statements to affect both how and where the output appears.

PRINT separators

The simplest of these are the punctuation marks known as PRINT separators. The function of each of the separators is quite simple: commas give you printing from the next tab stop, and semicolons give printing from the last print position.

If you use a separator at the end of a PRINT statement instead of between two PRINT items then it will have an effect on the next item to be printed. For example:

```
10 FOR I=1 TO 100
20 PRINT I, I*I
30 NEXT I
```

will print I and its square on each line, in regular columns, but the program:

```
10 FOR I=1 TO 100
20 PRINT I;I*I;
30 NEXT I
```

will print out a mess of numbers all over the screen. The effect of a print separator is cancelled by a blank PRINT line which automatically sends the next piece of output to the beginning of the next line.

SPC function

In addition to the separators mentioned above, there are also two, useful functions which can be applied in PRINT statements. The SPC function skips from the current cursor position the number of spaces specified. This is useful in that it does not print spaces, it skips over them, thus allowing you to print over part of a display while leaving other parts intact. Use PRINT 'spaces' to overwrite part of the screen with blanks. However, when using SPC be careful not to leave a space between the word SPC and the open bracket as this will make the computer think that an array SP has been declared. This would obviously confuse it.

TAB function

The other function you can use in PRINT statements is TAB. This allows you to select the character position (on the current line) from which you want printing to begin. Enough spaces will be skipped over to get you to that position, but remember that if the function would have to backspace then

a new line is chosen. The TAB function is particularly useful if you are setting up tables of information to be printed, and can save you a lot of time and effort working out how far to skip before printing begins.

The Commodore 64 lacks one function available on most other home computers and that is the facility to print at an absolute point on the screen, using a statement like PRINT AT X, Y. This can be done, but requires you to POKE screen memory rather than PRINT directly (see below).

When using any of the PRINT separators, or the SPC and TAB functions, be careful that you are using the ones you want, in the right places. The separators are particularly difficult because it is easy to confuse a comma with a full stop, or a colon with a semi-colon. Punctuation is taken very seriously by your Commodore 64.

POKE

POKE can also be used to push information onto the screen. Since the screen is mapped directly from memory locations 1024 to 2023 you only need to change the contents using POKE, and you can make things appear on the screen. You can print a character at an absolute X, Y co-ordinate on the screen by POKEing its ASCII value into memory using the following statement:

POKE (1024+(40*Y)+X), ASCII value of character

On the whole, however, most programmers will be better off using the PRINT commands which are easier to manage, and can be programmed more simply.

The most use of POKEing to the screen will be made by machine code programmers. They will have to do all screen output by pushing information into screen memory. POKEing to the screen is also widely used when you start dabbling with colour, and graphics, and with all other uses of the VIC II chip. These uses will be covered in subsequent chapters.

OTHER PERIPHERALS

The same applies here as applied to input in the previous chapter. Most peripherals will be very easy to use, and will either have hardware or software built in to help you use them without having to do any serious output programming. Certainly the more widely used peripherals like printers, joysticks and modems should not present any problems.

You may, however, get incomplete compatibility with some peripherals which claim to work with your Commo-

dore 64. Naturally, this is less likely to happen with Commodore's own peripherals than with those from an independent manufacturer, since they will have been specifically designed to work with the 64. On the other hand, you may find that you are offered better features on the independent products, and in some cases (like analogue speech synthesis) you can only buy an independent product.

As usual, you should see the peripheral in action with the Commodore 64 before you buy it. Peripherals are usually a lot more expensive than shoes — and you would hardly buy shoes without trying them on, would you? The difficulty of programming peripherals to work with the Commodore 64, when they are not directly compatible, should not be underestimated. It is almost always possible, but the cost in time alone could be excessive.

7

GRAPHICS

Graphics is one of the most interesting areas of computing, and probably played a large part in the fact that you own a Commodore 64 rather than any other home micro — unless you are lucky enough to have more than one! Computer graphics fall into two distinct areas: those that move, and those that do not. This chapter covers stationary graphics on your 64, while the next covers movement.

VIC II

However, although graphics fall naturally into two areas they are all controlled and defined by a dedicated chip inside your Commodore 64. The 6566/69 VIC II (video interface chip) is one of the most powerful chips ever to be put into an 8-bit micro, and controls the signal which eventually appears as a picture on your TV screen or monitor. By definition, then, the VIC II handles colour, images and, perhaps most importantly, sprites (for more on sprites, see Chapter 8). It has a certain amount of its own processing power, it can handle and interpret interrupts, and it can achieve one of the more complex features needed by a graphics chip — direct memory access (DMA). DMA gives a chip the ability to access and move large chunks of memory without affecting the main processor, and this is obviously essential when transferring information from memory to the screen.

These features mean that the VIC II is virtually independent of the 6510 main processor, leaving that to do the routine tasks such as looking after the keyboard, and to handle the calculations needed inside a computer as complex as the Commodore 64. The only sign you will see of an interruption is when the cassette unit is being used or in some of the more technically difficult I/O operations. When this happens you will see the screen go blank for a while (for example, when loading programs).

STATIONARY GRAPHICS

Before going on to look at the pictures you get with graphics it is important to look at the different ways in which the screen can be used. On some computers there are basically

two modes of operation: *low resolution* (character graphics) or *high resolution* (pixel graphics). The Commodore 64 has these, but also has the refinements of different amounts of colour to contend with, as well as the difference between having the character set accessed from ROM (standard) or from RAM. Altogether there are no less than eight graphics modes on the 64. Here is a list of them:

The eight graphics modes

Resolution	*ROM or RAM*	*Mode*
Low	ROM	Standard
Low	ROM	Multi-colour
Low	ROM	Extended Background
Low	RAM	Standard
Low	RAM	Multi-colour
Low	RAM	Extended Background
High		Standard
High		Multi-colour

In addition to these modes you can have several other variations on the state of the screen. You can define the screen to be 38 columns wide as opposed to the normal 40, and/or 24 rows deep as opposed to the normal 25. These dimensions would only really be used in applications such as smooth scrolling, since under normal circumstances you want the screen to be as large as possible, for maximum visual appeal.

Before going on to look at the problems encountered with stationary graphics it is worth pointing out that although you can create spectacular pictures from BASIC with the Commodore 64, you can do a lot more from machine code. Machine code increases the flexibility of the machine dramatically, but this is at the expense of your time. To create the really impressive displays you want from machine code will take an enormous amount of time and energy. They will, however, appear faster than anything you could hope to achieve from BASIC alone.

Machine code graphics

Although machine code graphics programming is complex, you will find that even simple BASIC graphics causes lots of problems. These will frequently be related to the fact that graphics are a highly subjective medium. Only you know

whether you are happy with the results of a particular graphics program. And only you know whether you are going to take the plunge, and dive into machine code, if you are not satisfied with the results from BASIC, which can be really good, despite what was said about the extra freedom, speed and flexibility brought by machine code.

COLOUR

The first area of stationary graphics to look at is that of colour. Colour is naturally fascinating (look at the impact of colour television) and it was partly because of the Commodore 64's high-quality colour, when it arrived on the market (along with its favourable price), that it sold so well. Until computers like the Vic 20 and Spectrum appeared there were very few colour computers around, and even now, with a much larger number, there are few which can compete on colour quality with the Commodore 64 (though some Atari models offer 16 colours from a palate of 256).

Standard mode

As you know, the Commodore 64 has got 16 colours. However, since some of these are black, white and greys it could be argued that there are rather less. Nonetheless, since black, white and grey play a large part in the creation of images from real life they are useful choices to have.

Each colour on the Commodore 64 has a number associated with it, and the colours are coded from 0–15. One result of this is that if you change the least significant four bits of any of the colour registers or any location in colour RAM then you will change the colour.

Colour RAM

Because colour RAM mirrors the screen memory (in low resolution modes) you can effectively POKE any screen location with any colour you like, and this will alter the colour of the character printed at that screen position. Remember that you can also change the colour currently being used for printing characters by printing one of the colour keys (on keys 1–8) to the screen, or by printing the CHR$ of one of the codes for those keys. For general applications it is often easier to do this than it is to fiddle around with screen memory, changing the parts you want to the colours you want.

Border and background registers

You can change the border or background colours very simply by POKEing the relevant colour registers, which are held at locations 53280 and 53281. As for colour RAM, it is only the least significant four bits (0–3) which affect the colour (range 0–15). The top four bits are not used in either register. I would have expected these bits to be used in colour RAM to show the background colour for each screen position, but obviously the designers elected for the system you have before you.

Given a detailed knowledge of machine code and the computer's operating system, you can see that there is room to implement this feature, and with it you could have much more colourful screens than those available at present.

One problem imposed on you in standard mode graphics is that although you can have all 16 colours on the screen you can only have two colours per screen location (ie within each 8×8 pixel square). If you want two lines of different colours to cross one another then you are going to have problems. However, the problem can be surmounted, at a cost, by the use of multi-colour mode.

Multi-colour mode

Multi-colour mode allows you to have four colours per screen location instead of the usual two, though there are two drawbacks. The first of these is that the screen resolution (normally 320×200 pixels) is reduced horizontally to 160 double-width pixels, and the second is that you are restricted to using the first eight of the 16 colours, since there are now only three bits instead of four in which to store them.

The fourth bit is now used as a sort of multi-colour flag. If multi-colour mode is being used then when bit 3 is set to 1 (ie the colour was previously in the range 8–15) that character is displayed in multi-colour mode. If it is set to 0 (colour previously 0–7) the character is displayed normally. This allows you to have a screen of mixed multi-colour and standard mode characters, which is a useful facility.

Multi-colour register

To switch multi-colour mode on and off the two following commands are used. These simply set and reset bit 4 of location 53270, one of the colour registers. To switch on, use:

```
POKE 53270,PEEK(53270)OR 16
```

To switch off, use:

POKE 53270,PEEK(53270)AND 239

Multi-colour mode looks first at bit 3 of the colour RAM to see if it is set, and if so, it prints the contents of screen memory for that location onto the screen in multi-colour mode. Since the pixels are now double-width original pixels, the computer obviously needs decision criteria to help it get the colours right. With each pair of pixels there are obviously four possibilities for the resulting colour: background 0, 1 and 2, and the colour specified by the three lowest bits in colour memory. These are chosen by a combination of the bits in screen memory as follows:

00	gives background colour **0**	location 53281
01	gives background colour **1**	location 53282
10	gives background colour **2**	location 53283
11	colour from lowest 3 bits	locations 55296–56295

One of the implications of this particular feature is that you can instantaneously change every associated multi-colour pixel pair simply by flipping the contents of the two registers 53282 and 53283. Doing this in sequence is a very effective way of getting a screen to flash.

Switching modes

Something to remember when using multi-colour mode is that it ought to be switched off before returning from a BASIC program. The problem is that because the pixel dots are looked at in a different way in multi-colour mode then multi-colour characters are very different from ordinary ones.

The net result is that program listings and immediate mode commands are virtually unreadable in multi-colour mode. The addition of the switching off command, setting bit 4 of the register at 53270 to 0 avoids the problem altogether.

Although multi-colour mode is a powerful feature of your computer, in practice it is time-consuming and fiddly to use. If you are intent on an impressive graphics display then you will go to the trouble to use multi-colour mode, but to get the most from it you will need to become adept at designing your own characters, since the standard set is no longer of much use to you.

Multi-colour editor

If you are planning to do a lot of multi-colour work then it may be worth your while investing in a multi-colour editor. This

clever piece of software is easily obtainable and will help you in any case with creating other user defined graphics. It is sure to save you a lot of time working out which bits should be set and which should be off.

Extended background colour mode

You will frequently find in graphics programming that using a single background is not enough, even though you can use the colour RAM to alter the colour of the character. To give you that extra flexibility the designers of the Commodore 64 have added the extended background colour mode feature.

This is easy to use, as long as you remember that only the first 64 characters can be used. What happens is that each character can now take one of four background colours; the foreground colour will still depend on the colour RAM.

Extended colour mode and multi-colour mode cannot be used together since the same colour registers are used for the two features. Extended mode also uses one more register (53284), for the third extra colour needed.

Switching extended colour mode

To switch extended mode into operation you have to adopt a similar procedure to that used to switch multi-coloured mode on and off. This time bit 6 of register number 53265 must be set for extended mode and reset to switch it off. Use the following instructions:

POKE 53265,PEEK(53265)OR 64 — to switch on

POKE 53265,PEEK(53265)AND 191 — to switch off

To get the best from the extended colour mode you only need to remember the way in which the extended colours are related to the registers and the screen codes of the characters being used. Each of the characters 0–63 can be used in one of four ways, the resulting screen code being in the following ranges:

0–63	register 53281	background colour
64–127	register 53282	extended colour number 1
128–191	register 53283	extended colour number 2
192–255	register 53284	extended colour number 3

Using graphics to redefine characters

In a similar fashion to the way in which colours were flipped in multi-colour mode you can flip the registers here to produce highlighting or flashing effects. Extended mode can be widely and effectively used within games, especially if applied with user defined characters from RAM rather than the standard characters from ROM. Again, however, remember that you can only have 64 of these characters defined out of the full set of 256, since the two most significant bytes are used to define which of the four background colours are to be displayed for that character (ie numbers 0–63 have bits 6 and 7 set to 0, numbers 64–127 have them set to 0, 1 etc).

In any of the modes described above the block graphics characters can be invaluable, and are a considerable asset of the Commodore 64, allowing you to create full screen images much faster than might otherwise be possible — especially when you consider the lack of standard facilities on the Commodore like line and circle drawing.

The last two graphics modes available on your 64 are the two which give you the most control over what actually appears on the screen — the high resolution modes.

HIGH RESOLUTION

At quite an early point during your use of graphics programs you will find that the programming of 8×8 pixel areas of the screen does not give you the accuracy you require. This is the time when you want to move onto a bit mapped or high resolution screen. Instead of printing characters from memory onto the screen, bit mapping relies on storing all the screen information in a table — all 64,000 pieces of it! Even though it only takes one bit to store one pixel this is still going to take up 8k of memory, and for this reason bit mapping is not perhaps used as often as you might expect it to be.

Unfortunately, plotting points on the screen in high resolution is a slow business on your 64 and the only way you can get round this is to use machine code instead. This is why most games are written in machine code — they would just be too slow without it. Bit mapped or high resolution mode can be used in both standard and multi-coloured mode, with the latter reducing the resolution, as you would expect, to 160 pixels across by 200 deep, instead of the normal 320.

Switching on and off

High resolution is switched on by setting bit 5 of location 53265 and switched off by resetting it, using the following pair of instructions:

POKE 53265,PEEK(53265)OR 32 — to switch on

POKE 53265,PEEK(53265)AND 223 — to switch off

Bit map

To set the multi-coloured bit map mode you must also set bit 4 of location 53270 to a 1 as before. There is no problem using the two together, but remember that the resolution is now halved and multi-coloured mode uses the screen information in a very different way from standard mode. You cannot use extended background mode with the bit map mode.

One problem which commonly occurs with high resolution is that you can have the BASIC program interfering with the high resolution screen. This depends, of course, on where your BASIC program currently resides, on its length and also on the positioning of the high resolution screen. For more information on memory management and hints as to the positioning of graphics and programs see Chapter 10, which deals with exactly these sorts of problems.

Before using high resolution you must decide where to put the screen. This is done by setting bits in the register at location 53272, of which more in Chapter 10. A good starting position can be achieved by using the statement:

POKE 53272,PEEK(53272)OR 8

which places the high resolution screen at location 8196 and those following. Note that to clear the screen you cannot now use the standard CLR instruction. Instead, you have to POKE the 8,000 locations concerned with zero. This can be done in a simple loop, but takes time.

Placing the screen in memory

Once the screen is set up you can POKE the relevant bit into memory to switch on the corresponding bit on the screen. Be careful when working these out, as the screen is laid out in characters, as before, only this time each character is made up of eight bytes.

This means that the bytes across the top pixel line of the screen are actually bytes 0, 8, 16, 24 etc. This may seem complicated, but the screen has been organized this way so

that you can have mixed modes — split screens with part being in low resolution and part in high. To do this you need a machine code routine which monitors the interrupts, and can use these to time the switching on and off of high resolution. The resulting screen display will be partly in high and partly in low resolution.

Using sprites as windows

For all practical purposes, you will probably find it easier to use sprites as high resolution windows rather than trying to monitor interrupts. For one thing, the sprites are easily accessed from BASIC while the interrupts require you to use machine code. The screen layout of the bytes in blocks of eight also helps if you want to have some characters in multi-colour mode and some in standard, as you would expect.

The 8×8 pixel squares

In high resolution mode colour RAM is no longer used to define the colours of the areas of the screen. Instead, the previous area reserved for screen memory (1024–2023) is used as a colour bank for each 8×8 area on the screen.

The top half of each byte is used to store the colour of the pixels set to a 1 and the bottom half is used to store the colour of those bits set to a 0. As before, standard mode allows two colours in each 8x8 square, while multi-colour mode allows four.

Altering the boundaries of the 8×8 pixel squares can only be achieved by using complex machine code. The idea is to convince the Commodore 64 that the character borders can move, usually up to four pixels in each direction. Printing in the new 8×8 pixel squares will change the character borders, and therefore the areas in which the colours are constant. There are also several other acceptable methods, but they all require a detailed machine code program to implement them.

Using variables to store colours

One tip worth noting is that if you are experimenting with different colours in different areas of the screen then you may want to declare the colours as variables at the start of the program. This way you can alter all of any one colour simply by changing the variable assignment at the start of the program, rather than having to search through the whole thing trying to find the assignments which used that particular colour.

Even if you do not use this method in the final version of the program it can be well worthwhile doing so when you are developing ideas, and it can give you the extra flexibility you want when designing and developing ideas.

DRAWING

The Commodore 64 falls below many other micros in its lack of drawing routines. This is compensated for to some extent by the block graphics which can be used in low resolution mode but the lack of even the simplest routines to draw lines and circles makes the creation of high resolution pictures both more difficult and much slower than it ought to be.

Pixel co-ordinates

To draw high resolution pictures using lines and shapes you must first work out the pixel positions in the 8K of memory which you want to change. Doing this obviously involves masking out the parts you do not want to alter, and using the AND and OR functions to change the bits. Because POKE and the logical functions are not especially fast, and because there are so many pixel positions on the screen (64,000), any routine you write in BASIC will be extremely slow with longer applications taking minutes rather than seconds.

Unless you are prepared to write routines in machine code which can be called from BASIC, or to write all your programs in machine code then there really is no solution to this problem. Computers like the Spectrum get round the problem by having extra BASIC keywords which plot the points in sequence for you, giving you features like line drawing as standard. Obviously within the operating system these are simple machine code routines which you call, and it seems a pity that the designers of the Commodore 64 stinted by not providing these functions.

You will never, in practice, be able to create a picture one dot at a time, Instead, you will have to use a system of equations called iteratively within loops to create the best effects. It is helpful to know the formulae of some of the more simple functions like a circle (X*X+Y*Y=R*R), though functions like a straight line are somewhat easier to implement.

One advantage of the method of high resolution implementation is that you can 'look at' the screen simply by examining the right area of memory. This is obviously helpful when programming games, though the need to do this is somewhat reduced by the flexibility of the sprites, and in

particular by the sprite collision mechanisms built into the 64.

Remember in any high resolution drawing that the colours are still effectively in low resolution — on a character basis. As long as you see the difference between the pixel drawing and the character colour positions then you should be able to get the best from your Commodore 64.

CHARACTER IMAGES IN MEMORY

On the 64 you have a wide range of characters available from ROM. There are 256 of these, and they include the alphabet, numbers, punctuation and block graphics. However, there are often occasions when you cannot get all you want from these characters alone.

Redefined graphics

One of the best ways of getting effective graphics is to transfer some or all of the character set into RAM and then redefine them for your own use. They can be easily called up, and are obviously a lot faster to use than a high resolution screen since they can be printed up directly onto the screen without having to be POKEd into memory, bit by bit. Defining your own 8×8 graphics characters is relatively easy, and they will enhance almost all programs.

As you probably know, the redefined characters have to be placed in their own character matrix, the address of which you can choose yourself (see Chapter 10 for more details on the positioning and moving of memory blocks). It is usually just as well to copy one of the two ROM character sets down into RAM so that you have some characters to play with and some which you will want to redefine.

Since each character is 8×8 pixels it takes 8 bytes to define each character, and this is laid out as follows (for the letter A, for example):

1	00011000	=	24
2	00100100	=	36
3	01000010	=	66
4	01111110	=	126
5	01000010	=	66
6	01000010	=	66
7	01000010	=	66
8	00000000	=	0

Defining the graphics can be a time-consuming process, but if you set up eight DATA statements then you can go through them on screen putting in 1s where you want ink and 0s where you want paper. The lines can be edited on screen, until you have the shape you require. Once you are happy you can run a program to POKE the values from the DATA statements into memory at the correct position.

Example of character redefining program

When the character has been defined you can note down what the eight numbers worked out to be (24, 36, 66 etc, in the above example), and then code these into shorter data statements, rather than having to read in all of the ones and zeros in the string. However, do not do this until you are sure that the graphic is just the way you want it. The following example shows how easy it is to redefine a graphics character:

```
 10 DIM T(8)
100 FOR J=1 TO 8
110 LET T(J)=0
120 READ A$
130 FOR K=7 TO 1 STEP -1
140 LET CH$=LEFT$(A$,1)
150 LET A$=RIGHT$(A$,K)
160 IF CH$="1" THEN LET T(J)=T(J)+(2 K)
170 NEXT K
180 NEXT J
200 FOR L=1 TO 8
210 POKE 8200+J,T(J)
220 NEXT J

1000 DATA "00011000"
1001 DATA "00100100"
1002 DATA "01000010"
1003 DATA "01111110"
1004 DATA "01000010"
1005 DATA "01000010"
1006 DATA "01000010"
1007 DATA "00000000"
```

This particular example would poke the defined character A into the second position in a character block starting at 8192. Obviously you could choose any position in which to store the final result, as well as any data you wanted for the redefined character. On its own this character will not do anything, but it can easily be displayed by calling the second character in the matrix. Redefinition is very simple

once you have all eight DATA lines set one above the other like this. If you print the contents of the array T, above, you will find that these numbers can be put into a DATA statement which will save you the trouble of converting the 0s and 1s into real numbers.

Using REM when redefining characters

When redefining characters it is always a good idea to add a REM into the program explaining what the graphic is, and the position in which it has been stored. If you forget to do this then you are likely to forget what is what and you may think that you have lost the graphic you wanted. Moreover, if you want to further modify any characters, at a later date, it is helpful to know, for example, what exactly character 34 was.

If you want to, you can code the user defined graphics yourself (this is what the above program does, after all), by adding the bits together which are set and not set, but it is recommended that you use the program to do this part of the work for you as it is probably less fallible.

With care you can build the graphics characters together, to make larger graphics than those which fill one character space, though sprites will probably prove to be the best answer if this is what you are aiming for. Ingenuity also plays a large part as you can use repeated patterns of your own characters to create the illusion of more versatility than actually exists. Since these redefined characters can be printed quite quickly onto the screen you can start having animated games, even before you start using sprites (see the next chapter). However, before looking at movement there is one more area of block, user defined, and high resolution graphics worth mentioning: stored graphics.

Stored graphics

You can redefine as many characters as you are likely to need, but these are small, and obviously not as impressive as a large high resolution display. However, large displays are slow to produce, and the wait is often not worth the result. What you can do to avoid the problem is capitalize on one of the most important features of your computer — its large memory. You can save whole screens or areas of screen to memory and then call these back when you need them. These screens can be used as fresh backgrounds to your sprites, once they have been designed.

The snag with these screen saving and reloading ideas is that they are hard to implement from BASIC. However, there

are some programs around, both commercial and in magazines, that will help you to do this.

The best results are going to come from machine code programs, where whole blocks of memory can be mapped into the high resolution screen memory, giving you very fast results.

Once you delve into the realm of saving whole screens you start to find out about the problems of space. Whole screens take up 8K of memory, so by the time you have taken away the BASIC program, and the operating system and so on you will be lucky if you can get even four or five screens saved. Like most problems on the Commodore 64 this one has its solution rooted away in the depths of machine code. The answer is the screen compaction routine.

Screen compaction routines

Screen compaction routines are now quite common, though when the Commodore 64 first appeared they were hard to find. The routine has to be in machine code, but you should be able to find one which you can call from BASIC, so that even if you are not yet using machine code you can still make use of it.

The idea behind the routine is simple. In most drawings or pictures not every pixel is different, and therefore if you go down the screen line by line you will find whole areas that are ink pixels or whole areas that are blank. What you can then do is code, say, 184 dots in blue as two bytes, giving the colour (blue=1) and the extent (length=184). If the blue line ended with 30 pixels in red then you could simply code that as two more bytes (2,34).

In this way a whole line which might normally take 40 bytes (320 pixels, 8 pixels per byte) might be reduced to 4 or 8 bytes. A good screen compaction routine should be able to save you about ninety per cent of the space you would use without it. Of course, a screen compaction routine is not much good without another routine which decompacts a compacted screen. With an efficient compaction and de-compaction routine you should be able to store about fifty different simple screens, and you will have seen games on the market which do just this.

Graphics design packages

If you want to write graphics games then it is certainly worth investigating the possibility of getting a graphics design package. These packages are not as expensive as you might

expect, and provide not only the wherewithal to create screens which can be used as backgrounds, but also to compact them, store them where you want and retrieve them in the minimum amount of time.

If you are very enthusiastic then you might consider writing this package yourself, but you will find yourself involved at the heart of machine code, if you are going to get the kind of easy to use package you want. On the whole, it is probably better if you buy the package and then use it to design original screen layouts, rather than spending your time writing routines that have already been written by someone else.

The best games are going to incorporate most of the ideas outlined here, but are also going to have the one graphics device not explained here: movement.

8
ANIMATED GRAPHICS

Most people would agree that graphics, colour and movement (animated graphics) are all features of the best arcade games. This is reflected by the fact that now there are home computers available with sprites, with a wide range of colour and other graphics features built into them, where previously there were less of each. Since the Commodore 64 was primarily designed as a games machine it probably has the most advanced built-in graphics facilities, though in typical Commodore fashion you have to dig around in memory, changing single bits here and there, to access these effects.

Although there is naturally a certain amount of difference between the type of movement you see in an arcade game and that you can effectively implement on your Commodore 64, the results you can achieve with sprites are probably of the highest quality available on home computers. This is all thanks to the VIC II chip, which manages your eight sprites on screen, as well as giving you bonuses like screen scrolling, collision detection, sprite priorities and multi-coloured sprites. The VIC II is a very powerful chip.

THE ILLUSION OF MOVEMENT

Starting from the beginning, however, it is important to realise that all computer movement is an illusion — unless you move your monitor around the room! The same applies to many other areas which you think of as movement too, film being one of the best examples. Film works by showing you so many still images in succession that you gain the impression of movement.

Most importantly, the illusion is convincing, and this is what you should be aiming for, and what, with sprites, you can achieve.

Using PRINT for animation

You can get primitive movement on your computer by using the old (and obvious) trick of printing something on the screen, wiping it off and then printing the same thing one character position away from where it was before. This is what you will have used to animate your redefined graphics, and also in the general movement of text. As long as you

apply a bit of logic to the idea then you should not have too many problems with animating single characters, or even long repetitions of characters, as in the 'Snake' style of game.

The important thing is to remember to wipe off the previous state of whatever is moving before printing the next one. Using an extra space tacked onto the item to be printed is the most effective method, but remember to put this on the correct side of the character. If you do this you will now find that the PRINT position on the screen will have advanced too far. This problem can be avoided by embedding control characters into your PRINT statements (like delete), though doing this can make the program lines themselves difficult to edit. It is often easier to add the CHR$ codes to the PRINT statements, or even to code these characters as strings and add them in where needed.

To use this method of movement in all directions you will have to embed many characters and it is easy to add in the wrong ones, or the right ones in the wrong place. Exercise caution, and try your ideas out and you will soon find your redefined characters moving about the screen with ease.

Sooner or later, of course, you will move onto the added complexity and flexibility of sprites. But be warned — designing and programming sprites can be a wearisome business, though the results usually justify the effort. Even if you know what your sprite is going to look like you will have to type in the data for it, and this takes a lot of time — even with the Commodore's useful screen editing facilities.

BACKGROUNDS

Ideally, to make movement look really convincing, you need a background against which the sprite can be seen to move. In real life it is always more convincing seeing an aeroplane flying away from take-off than it is to see it against a clear blue sky. Remember that all the time your eye is going to try to map what it sees on the screen into some version of reality. If there is a background against which something is moving then you add in your own impressions of movement, speed and direction. A plane against a clear sky seems unreal because there is nothing against which it can be mapped — except of course the edges of the screen, the equivalent of a window frame.

Fortunately, the sprite facility of the Commodore 64 allows you to use any background you like, and so you can design elaborate screens, save them away, and then use

them as backgrounds for your sprites. On many other home computers, moving objects is difficult because they obliterate whatever was underneath them as they move. On the Commodore 64 all of this is handled by the fiendishly clever VIC II chip.

Importance of scale

One last tip about using backgrounds is this: there is little point in using a background, however elaborate, unless your sprite and background are to scale. Remember when designing backgrounds that your sprites are 24 pixels wide by 21 deep and that therefore anything you want to move must fit into that shape and size. However, to a certain extent, you can have different sized sprites. Smaller sprites can of course be designed within the larger (24×21) frame, and larger sprites can be made up of a combination of standard ones — a car for example might use two sprites joined horizontally. The mechanics of multiple sprites are not easy to program, but are nonetheless possible (using machine code, naturally).

SCROLLING THE SCREEN

There are many occasions when you want a sprite to remain still and the rest of the background to move. On many home computers this is difficult to achieve, but most of this feature is provided in the VIC II chip.

This principle of leaving one graphic in the centre of the screen and moving everything else is widely used in commercial programs, and works on the theory that if you are in the middle of the screen then you see things moving relative to you — the view from a bus window. This is the technique used in popular games like *Defender* and *Pole Position*.

In practice, you do not have to use a full-screen scroll to create the illusion of scrolling. Lines and images can be drawn and redrawn at different positions on the screen to save you the effort of having to move the whole screen image, though since the drawing of lines is a slow business from BASIC it really requires the use of machine code. This type of scroll can be achieved much faster than a full-screen scroll.

Full-screen scroll

However, compared to this partial scroll method, a full-screen scroll can look very impressive. The catch is that to do it you need to know intimately the inside of your

computer. The theory is quite simple: blank off one edge of the screen, place the new part to appear on one edge, and then move the whole screen one character across. The (visible) screen can be made narrower to 38 characters instead of 40, though both of the extra columns are still in memory. For vertical scrolling only one row is lost, giving 24 instead of 25 lines. Use the following commands to make the screen narrower:

POKE 53270,PEEK(53270)AND 247 to reduce to 38 columns

POKE 53270,PEEK(53270)OR 8 to return to 40 columns

POKE 53265,PEEK(53265)AND 247 to reduce to 24 rows

POKE 53265,PEEK(53265)OR 8 to return to 25 rows

Scroll registers

In practice, a full-screen scroll really needs a machine code routine to run it, since the process of putting new information into the hidden part of the screen and then moving everything along one character is very difficult to arrange. From machine code you can achieve tremendous effects using scrolling, but because of the nature of the hardware it is often easier to use sprites rather than scrolls for your special effects. The scroll will be smooth, as long as you have arranged the scroll registers correctly: these are the first three bits (0–2) of locations 53265 for vertical and 53270 for horizontal scrolling. Note that scrolling will only work with the reduced screen sizes.

MACHINE CODE MOVEMENT

The more sophisticated the movement you want, the more difficult it is to program in BASIC. Even when the facilities are directly available to you in BASIC (as sprites are), their best side is seen when they are used from machine code. If you do stick with BASIC you will find that you are now working consistently at bit and byte level, which is , in any case, no less complicated than programming in machine code.

If you are going to use the true power of movement on your computer you will have to learn machine code. To some extent you can avoid this by using ready-built machine code routines which you can hook into from BASIC. These can be passed the relevant parameters needed (sprite number, direction and so on), and you can sit back and watch the results.

Ready-built routines like this require no knowledge of machine code; you only need to know what parameters the routine is expecting. Routines like this exist for scrolling, fast sprites, windows and also for many of the graphics background functions mentioned in the previous chapter (shape drawing routines, fill and so on).

However, even with a whole library of these routines you are still going to need machine code of your own to get the precise effects you want. Do not be put off from learning machine code. If you have a sound knowledge of BASIC and a grasp of the fundamentals of the ideas behind programming languages then you should not find machine code too difficult. On the Commodore 64 it is easier than on most machines, as to get anywhere, even in BASIC, you will have had to come to terms with the way the bytes inside the machine operate. This is necessary to an even greater extent in machine code, but you will have had an excellent grounding simply in using the VIC II registers.

There are some very good assemblers and debugging tools available for the Commodore 64, as well as several books on the subject, and so you will have plenty of backup when you run into problems. The results you will get from machine code programming are usually rewarding enough for the effort you have to put into it, and one thing is certain: in BASIC you are never going to get the kind of smooth movement and fast graphics you can achieve with machine code (See Chapter 12 for more details of using machine code).

USER CONTROLLED MOVEMENT

Some of the problems you will find with movement, whether in BASIC or machine code, come when the movement is under the program user's control. This situation has to be organized for any interactive game, where the user moves something around the screen, as otherwise they will not bother to play the game. It is the prospect of interaction with the machine which prompts people to put money into games machines in the first place.

On the Commodore 64 you can use the GET command to tell you when a key has been pressed, and you can look out for the specific keys you want to input (cursors, fire keys etc) with the same command. The joystick ports can also be monitored, and in practice most games will be best played using these rather than the keyboard. It is important while monitoring the keyboard to make sure that you have

repeating keys if you want them, or that the repeat is disabled if you want the user to press the fire button every time, rather than allowing it to be continuous. The same, naturally, applies with joysticks.

As long as decisions like these are made in advance then you should be able to control the way the game is played with little difficulty. It is often only when you change your mind at the last minute that you have problems.

You may also have problems with the positioning of redefined graphics characters or sprites on the screen. It is best to have this positioning as a separate part of the program, so that it can be programmed as a self-contained unit. In this way you can develop it on its own, and have the freedom to experiment with it and with various positionings on the screen before integrating it into the rest of the program.

SPRITES

Although sprites may at first seem daunting they are in fact quite simple to use, if time-consuming to program. As long as you keep your head above the water then you should not experience too much difficulty with them.

Sprite design

The initial problems you will experience will be in designing the sprites in the first place. The simplest solution really is to acquire a sprite designer, which allows you to design the sprite on-screen, editing it with the cursor keys. These editors or designers are available from magazines, books, and commercially, and choosing one is simply a matter of personal preference.

If you really want to design sprites yourself then you have two options. One is to use an extended version of the program in the last chapter, where you edit DATA statements on screen.

All you need to do is to increase the size of the array, T, to 63 long, and the length of the strings being read in to 24. In addition, you have to split each string into three shorter strings of length 8 each, as they are read in — these can then be treated in exactly the same way as they were for redefining characters. If you are really feeling masochistic then you can of course do all of this by hand. Draw the sprite you want into a 24×21 grid, convert each section of eight bits into its byte and type the numbers into DATA statements. It

really is much easier to get a professionally written sprite designer or editor.

Although you can have eight sprites on the screen at once it is well worth remembering that you can have up to 64 sprites designed and waiting in memory. These can then be accessed by moving the sprite pointers (held, along with most of the other information you are likely to want or need in the VIC II registers). It is this technique which will be used when you come to animate your sprites within their space as well as across the screen (see below).

VIC II registers

It is important to keep a track of the state of the VIC II registers. These are accessed in the 6510 from locations 53250 to 53294 (locations 0–46 in the VIC II), giving 47 registers in all. There are actually substantially more than 47 pieces of information stored within these registers as in many of them it is the bits (and not the whole byte) which are significant. As long as you keep a clear head then the VIC II registers can only be of use to you. Remember, while looking at single bits in these registers, that AND wipes things out and OR adds them back in.

Sprite table

Remember, too, that the sprites are stored in a sprite table, after the end of the screen memory. If you should move the screen memory (and there are many reasons why you should want to do so — see Chapter 10) then it is a good idea to move the sprite table with it, to keep things as tidy as possible. If you do this then you can be sure that your sprites will not be corrupting any other data in the system, or that a high resolution screen is not overwriting a sprite table.

Sprite collisions

You should also take care when using several other 'hidden' features of sprites on the Commodore 64. The VIC II notes whenever two sprites collide or when a sprite collides with a character, but it is up to you to take any necessary action as a result of this. The two registers at 53278 and 53279 respectively are used to make a note of collisions between sprites and between sprites and characters. If bits 3 and 5 of location 53278 are set after two sprites have been used then you know that sprites 3 and 5 collided at some point. The trick is to find out where, since the bits being set only tell you that at some point the sprites did collide.

To find out where and when sprites collide (with each other or with characters) you must monitor the bytes within a short loop. In BASIC this will inevitably result in an overlap because the loop is too slow for the sprite — by the time you notice that a collision has taken place the sprites will have overlapped.

To be effective, the collision detection routine needs to be in machine code, like just about everything else which is to be effective with sprites or fast graphics.

The last thing to remember about collision is that the bits which get set by a collision are not automatically reset once you have read them. If you want to detect first one collision and then a second you must clear out the collision bytes between tests.

Sprite priorities

There is a register at location 53275 which has a bit for each sprite to define whether the sprite is in front of or behind the background picture on the screen. If a sprite's bit is 1 in this register then it appears behind the background data. If the bit is a 0 then the sprite has a higher priority than the screen, and so it appears in front of it.

In addition, a sprite's number (0–7) defines how it crosses other sprites. Sprite 0 appears in front of all the other sprites, sprite 1 appears in front of all the other sprites except sprite 0 and so on. This priority feature of the Commodore 64 allows a real three-dimensional effect to appear in your programs since objects can be seen to pass in front of one another. Used with care the collision and priority features should present you with few problems.

Switching sprites on and off

Little could be simpler than the actual process of switching the sprites on and off. There is an enable register at location 53269, and switching the sprite's numbered bit to a 1 turns the sprite on (making it visible). Switching the bit to a 0 turns the sprite back off again. The one problem you can find is that sprite 3 is the fourth in the table (0–7), so switching sprite 3 on requires the instruction:

```
POKE 53269,PEEK(53269) OR 8
```

where you might expect to use

```
POKE 53269,PEEK(53269) OR 4
```

Sprite colours

Each of the sprites can have any of the 16 standard colours, and the colour is stored in registers at locations 53287 to 53294. The sprite ink pixels will be in the colour specified by the number in the register associated with that particular sprite. Remember that any pixels not 'inked' in, in the sprite, will show whatever is in the background picture beneath them.

Multi-coloured sprites

Although the prospect of multi-coloured sprites may seem alarming, these are simple enough to use as long as you think of them as being like any other multi-coloured part of the screen.

Multi-coloured sprites are not 24 pixels across, but 12 bit pairs across, and the pairs are used to specify the four colour options as follows:

00	transparent (ie the screen colour)
01	sprite multi-colour number 1 (53285)
10	sprite colour register (53287–53294)
11	sprite multi-colour number 2 (53286)

As for normal multi-colour, you must set the bit for each sprite to be multi-coloured to a 1. The register used for this purpose is at location 53276.

Expanded sprites

The Commodore 64 also allows you to expand sprites by a factor of two either horizontally or vertically, and subsequently to reduce them. This is done by setting and resetting the appropriate bits (depending on the sprite number) in the registers at location 53277 for the X direction and location 53271 for the Y direction. Setting the bits to 1s doubles the sprites, and resetting to 0s resets them to their original size. This feature is well worth utilising, though be careful not to make the mistake of thinking that the resolution has doubled — all you have is a larger sprite than you had to start with.

Incidentally, although this seems impressive (and therefore complex) the theory behind this kind of magnification is simple. To double any byte it is simply necessary to rotate it to the left by one bit. For example, the byte 00001100 (12) rotated left becomes 00011000 (24). Doing this to a whole sprite is not difficult, the hardest part being to get the carries taken across from one byte to the next in each row or column.

Sprite animation

The last area of movement is animation. Like the theory for sprites, the theory of computer animation is simple. To animate a character you simply need several different frames of the same image, with slight differences between each. When the images are shown in fast succession it gives the impression that the character is moving. Put this in a sprite and not only is the character moving, but it is travelling too — essentially the same idea as that used to make films. In films you cannot tell that you are watching separate frames of film as long as you are seeing at least fourteen frames a second.

Animating your sprites on the Commodore 64 is not especially difficult but is time-consuming because the data for all the different states of the sprites has to be typed in first. What you must then do is keep changing the sprite data pointer in rotation around the different frames of the animation. This causes the sprite to keep moving, but to change shape as it does so.

The advantage of animating your sprites is that it adds realism — it tricks the mind into believing. It is bit like the difference between seeing someone walking down the street and seeing someone cycling behind a wall so that all you can see is the body gliding along. The cyclist looks unreal, whereas the pedestrian looks natural, because of the translation movement and internal movement combined.

Importance of realism

In all computer games your aim is probably to make things look as convincing as possible. Animated figures often look more convincing than ones which do not change shape as they move. Things like boats or aeroplanes look realistic if they just glide across the screen because this is what they are expected to do, but a person gliding looks rather strange.

The way to avoid this is to animate the character in several frames, showing the various positions of movement in a cycle. For example, if you want to show a rabbit running then you would probably use the legs and body extended for one frame and the legs bunched up underneath the body as another. If these were displayed one after the other then the impression of animation would be achieved.

To animate a person walking across the screen you generally need three states of animation, or frames. Two show the figure with the legs apart; one with the left foot

forwards, the other with the right. The third frame show the two legs together, in the act of crossing one another. To make the animation realistic this third state should be shown between the other two, and at the end of the cycle too — in other words there are four states in a walking movement, although two of them (states 2 and 4) are the same.

Animation can look very disjointed or misplaced if the images making up the sequence are badly drawn (created). It is always a good idea to perfect the separate images before you set to work on animating them. Even more complicated are the problems which you face if you get the sequencing wrong. I once had a dog on the screen which rather paradoxically moved both its left legs together and then both its right legs. As you can imagine it looked very amusing, but hardly realistic. With just two of the four frames changed in the sequence the dog looked perfectly normal, strolling back and forth across the screen, where only moments before it had been lurching unrealistically.

9

SOUND

Sound is a feature, like graphics, colour and movement, which will brighten up almost any program. However, it is with sound that you will possibly experience your greatest problems to date — simply because the Commodore 64 has one of the most sophisticated sound systems available. This is controlled by an independent 6581 processor called the sound interface device (SID), which is virtually a full sound synthesiser.

SID CHIP

The main problem you will find is that no sounds can be created without POKEing values into the SID registers. There are 28 of these, working up from location 54272 in memory, and each controls a different part of the sound which comes out. In this way the use of the SID chip is similar to that of the VIC II, discussed in the last two chapters, but at least with the VIC II you could see things happening on the screen. If something is not quite right in the SID then there is often no way of knowing what the problem is. And sound is, on the whole, less tangible than graphics, so when you get a sound which is not quite right it can be hard to know how to correct the problem — unless you are a musician.

However, do not be put off exploiting the Commodore 64's massive sound power. You can use the SID not only for playing tunes, but also for all the other noises that you might want to put into a program, such as explosions, vehicle noises, crashes, space sounds and ascending and descending pitches, which can be used with a whole range of graphics, from bombs to helicopters.

Before going on to look at the complex way in which the SID is controlled it is a good idea to have a look at sound itself. After all, the SID is simply using the mass of variables you pass it to produce sound, and it is these variables which cause the problems.

ABOUT SOUND

Starting at the beginning, sound can be reduced at its simplest level to a vibration in the air. This is what you hear.

Each molecule in the air moves up and down when a sound wave is emitted from a source and the effect is similar to that you see when a cork bobs up and down on a wave in the sea — the cork stays where it is, but the wave moves across it.

There are two principal factors which affect what you hear: the shape of the wave, determined by the *frequency*, and the strength of the wave, determined by the *amplitude*. Other factors will also make a difference to it, including the type of wave. Pure notes are sine waves, others can be rectangular, sawtooth, triangular etc.

Sound envelope (ASDR)

Of course it would be very hard for the computer to generate these rather abstract forms, and it would be even harder for you to define all the parameters needed. What you can do instead is define the approximate shape of the outside of the wave, which is called the *sound envelope*. Doing this gives you quite a good approximation to a waveform, and can be controlled by four parameters, which are the attack, decay, sustain and release, or the ADSR of the sound wave.

The ADSR is controlled in the SID which allows you to specify the attack as the rise time of the note, decay as the time taken for the note to fall to a steady amplitude, sustain as the volume of this steady amplitude and release as the time for the note to fall from the sustain level to zero. One implication of this is that you must add in a loop to hold the note at its sustain level.

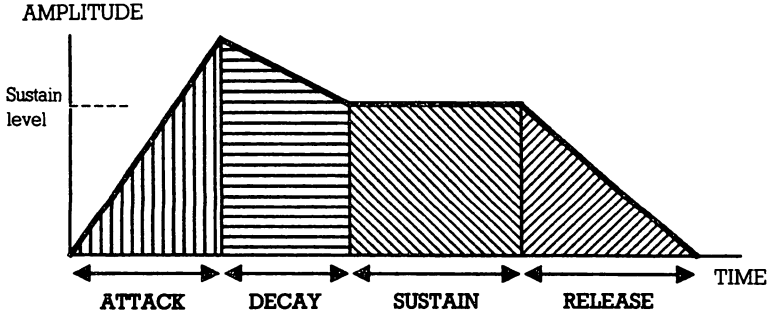


Diagram showing ADSR

Harmonics

The type of wave you use inside the envelope will affect the type of sound you hear. The reason for this is that each type of wave produces different harmonics. Harmonics are the extra frequencies you get when you play a note — natural

harmonics are 1, 2, 3 ... times the frequency of the fundamental note — and the reason different instruments sound different is that they have different mixes of natural harmonics.

USING THE SID CHIP

Moving on, have a look at how these ideas fit into the practicalities of the SID chip. The first thing to notice is that there are three voices, or notes which you can use. This sounds (and is) powerful, but it is worth realising that you cannot hope to mirror natural harmonics with this as you barely have enough voices to produce the fundamental notes. The range of these three voices is eight complete octaves (for musicians) or up to 4000 Hz (for scientists).

Main SID registers

To produce a sound on the Commodore 64 you have to go through a series of steps. Before describing these it is best to have a look at some of the SID registers you are about to use. Here are the registers for the first of the three voices, the other two are similar to this one:

54272 (00) — frequency register (bits 0–7)

54273 (01) — frequency register (bits 8–15)

The frequency is determined by a 16 bit register, and the low byte is stored first. Actual values which should be used for notes in scales can be found at the back of the *Programmer's Reference Guide*, for all eight octaves. It is important to remember that these values are not the frequencies in Hz, but in Commodore language. The relationship is:

$$\text{frequency (Hz)} = \text{frequency (64)} \times 0.059604645)$$

Since the frequency in the registers is affected by the clock inside the 64.

54274 (02) — pulse width register (bits 0–7)

54275 (03) — pulse width register (bits 8–11)

The pulse width register defines the amount of time a rectangular or pulse wave spends at the top and how much at the bottom. A value of 2048 will be a square wave, a value of 4095 will give continuous output.

54276 (04) . function control register

Bit 0 GATE: set to 1 to start attack, set to zero to trigger release
Bit 1 SYNC: set to 1 to synchronise with voice 3
Bit 2 RING MOD: set to 1 for non-harmonic sounds
Bit 3 TEST bit: can be used to synchronise voice 1
Bit 4 triangle: mellow flute-like sound (few harmonics)
Bit 5 sawtooth: rich brassy sound (many harmonics)
Bit 6 rectangular or pulse: change pulse width to change the harmonics
Bit 7 noise bit: if set to a 1 noise will depend on frequency bits

54277 (05) attack/decay register

Bits 4–7 select one of 16 attack rates (2ms to 8s)
Bits 0–3 select one of 16 decay rates (6ms to 24s)

54278 (06) sustain/release register

Bits 4–7 select one of 16 sustain levels (15 max, 0 min), note that sustain will stay at this level until GATE is set to zero
Bits 0–3 select one of 16 release rates (same timings as decay rates)

Tasks required when programming sound

Now that you can see the structure of the controlling registers you can see the sort of things that you need to do to generate sound on the Commodore 64. The best way of avoiding potential problems when creating sound is to draw up a list of the tasks you must do to get any sound at all out of the computer. A minimum list would look something like this:

- Set a variable up as SID (=54272)
- Clear the contents of SID (POKE SID to SID+28,0)
- Turn on the volume (POKE SID+24,15 for maximum volume)
- Select attack and decay (eg POKE SID+5,170 attack=10, decay=10)

- Select sustain and release (eg POKE SID+6,153 sustain=9, release=9)
- Select voice frequency (eg POKE SID+1,28:POKE SID,49. The resulting note is Concert A, 440Hz or 7217 in Commodore notes.)
- Select voice waveform (eg POKE SID+4,32 (sawtooth))
- Switch GATE to 1 (POKE SID+4,PEEK(SID+4)OR 1) (Note that the two commands above could be combined by the single command: POKE SID+4,33)
- Start delay loop to define length of sustain (eg FOR X=1 TO 250: NEXT X)
- Switch GATE back to zero, to release (eg POKE SID+4,PEEK(SID+4)AND 254)

And that is just for a single note, with one voice used out of the three! As you can see the programming of sound is an exceptionally time-consuming business, even when compared with graphics.

PROGRAMMING TUNES

The problems with sound come not so much from understanding what the SID chip does and how its registers are controlled, but from more intangible things like trying to get a tune to sound correct, or creating the right sound effect to go with an image.

The first hint with sound is that if you want to program a tune, most of the parameters can be set up in advance. The frequency and duration of the note (used in the sustain loop) can then be read in from DATA statements. In this way you can simply modify data rather than change the program structure to alter a tune.

Sound on your computer is one of the better advertisements for structured programming. If the data is put at the end of the program, and if variables are used for things like ADSR, then programming sound becomes much easier. For more on structured programming see Chapter 11.

Programming from sheet music

You can code a tune from sheet music into DATA statements relatively easily. To work out what the notes themselves are you can use the 8-octave table given in the appendices to the *Programmer's Reference|Guide*. The durations of the notes are quite easy to work out. Given that a whole note has a duration of 2048, the durations of the others can be derived from this.

The problem, of course, with coding a tune into DATA statements is often not so much in the conversion of sheet music but in the reading of the sheet music in the first place — and even if you can read music you may not have a score.

If you do not know much about the way musical scores are constructed then you will have to rely more on ear, and experimentation. Most of us can tell when a tune sounds wrong, without necessarily being able to read or write music. It is this skill that you must call into play if you do not understand the more subtle technicalities of music (like notes, scales, harmonies, melodies and the like). Although most of us like listening to a good tune we do not all know how to play one. Despite the SID being quite complicated to set up and use, the hardest problem confronting many of us is that of not knowing where to start with a tune.

One suggestion is to try playing the tune yourself, to get an impression of what it sounds like. However, if, like me, you are unmusical then this idea has no chance of working. Even if I did know a tune when I heard it, it would be very unlikely that I would be able to play it.

Trial and error method

If this is the case, then one of the best solutions is trial and error. Since the *User Manual* gives the note for concert A (POKE SID+1,28 for the high byte and POKE SID,49 for the low byte) then you can work above and below this quite gradually. After a while you should find that you can build up tunes from memory, just by repeated tries, getting closer all the time to the tune you wanted to hear.

Using a recording

If you find trial and error alone too difficult (and to be honest, I do), then you might find things easier if you get a cassette recording of the tune you want, and then work out the sequence of notes from that. I also use one of the Casio baby synthesisers, and get a friend to type in the tune, leaving me to try and program it into the Commodore 64. This method is

quite effective, as you can work on the 'this note is lower/higher than the preceding one' principle. Once you have got that far you can then adjust the tune until it sounds approximately like the original.

A taped or pre-programmed version of any tune is obviously going to be more useful than a record, as it allows you to play the same section of the tune over and over again, until you can imitate, and finally program it.

For the unmusical, writing tunes on the Commodore 64 is bound to be one of the most difficult areas of programming. Obviously if you already play a musical instrument then you will have a head start, but even if you do not, you can get surprisingly good results from trial and error alone.

Varying keys and speeds

One way of adding variety to a program is to use one tune, but in different keys and at different speeds. This can be done by putting the tune into a subroutine and then calling it after changing the values of variables which are used when the tune values are POKEd into memory.

PROBLEMS WITH SOUND

One problem which crops up quite frequently with sound is that while the program is going through the sound routine it stops doing other things. There are two things you can do to make sure that this does not bother the program user: one is to have a good display at the time the tune or sound is being played, and the other is to try and replace any pauses you had, or delay loops, with the sound routines.

If you distract the user with a visual trick just before a long tune then he or she will not mind the visual pause which accompanies the tune. In many instances you will find that many short sounds are actually more effective than one or two long ones, and if this is the case then they can be interleaved with movement and graphics. Short sounds and tunes will obviously need less time and programming effort too.

If you make sure that sprites are being used at the same time as a sound routine is being called then the problem of the visual pause will not affect you so much. This idea can be extended into sound as well as sprites by the use of machine code. Once you know how the computer's interrupts work you can program the sound from machine code, and you can have this routine interrupt-driven — unaffected by anything that may be happening in BASIC. You have probably seen

this used in commercial software, where a tune is playing the whole time that the game is running, regardless of what else is being done with memory or the screen. Needless to say, the programming involved in this is not simple, but the results can be well worth the effort.

Writing separate sound units

In line with the rest of this book, I would recommend that you write any sound routines or tunes as separate units. This is particularly important with sound, partly because of the amount of experimentation you sometimes need to get a tune right. Once you are happy you can then save the data for it on tape, ready to be used later on.

The tune or effect need not then be restricted to the program you are currently working on. You can use the sound routines as just another part of the ever-growing library of useful routines and programs which can be brought in when needed. Once you have been building things up like this for a couple of months you will find that using sound is simply a question of selecting the sort of routine you want and making fine modifications to it. No more the hours of experimentation you had to go through before.

When writing these routines it is therefore a good idea if you keep a note of the line numbers of the data statements involved. In practice, it is a good idea to do this with any program likely to be merged into another, as this can save you overwriting something important.

Although it is hard to write sound routines on their own it is better to do this than get sidetracked into it when you are in the middle of writing a game. It is often better to write the whole game first, and then see how it plays.

After doing this you will probably be able to see the places where it could be best enlivened by sound, and you can then write the routines with the game in mind. Alternatively, if you already have a good collection of sound routines you can just merge them into the program, with appropriate calls to them, where required.

FILTERS

There are other facets of the SID chip which have not been mentioned yet, but which can be used in refining the sounds or tunes you want to create. The two registers (at locations 54295 and 54296), after the three voices, are used for filtering the output from the voices. A combination of the mode

register and the RES/FILT register are used, and the effect, depending on the bit set, is to cut off high, low or medium frequency sounds. You will have seen this feature on almost all stereo cassette units, and the same principle applies here. There are those who believe that the refinement is an advantage and those who believe it takes something away from the sound. Experiment, and you will discover into which category you fall.

SOUND EFFECTS

Sound effects require a completely different approach. Since you are simply dealing with sound rather than music you can now start with any sound and experiment around it. Loops and very short duration escalating sounds are often the most effective, but you can create good effects with, say, one note going down and another one going up at the same time. Experimentation is the only way that you will find out. And since sound seems to be virtually compulsory in the games of the 1980s it looks like you will be experimenting for quite a while yet.

If you want to know more about using the SID chip to its best advantage then look out for the long promised book from Commodore: *Making Music on your Commodore Computer*. This is reputed to tell you far more about the true versatility of the SID than any other documentation. Since it comes from the makers you would expect it to.

One of the few serious failings of the Commodore 64 is that you are provided with no BASIC sound commands at all — you have to battle on with PEEK and POKE, probably the two commands you will use most in the whole time you program a Commodore 64. It is a problem which seems unnecessary, but since you can get round it, and since you cannot at this late stage alter the structure of the machine, it seems pointless to go on about it. Given time, you will produce superb sounds with the SID chip, and you will accept the way it works.

10

MEMORY

Everything you can do with the Commodore 64 relies on memory — otherwise none of the information could be stored and recovered. Your BASIC program, the screen picture, the colour RAM, sprites, variables, arrays, redefined graphics and the standard character set are all stored in memory. In addition, of course, the BASIC interpreter and the Kernal (the operating system) are also tucked away in memory. But there is still quite a lot of space left over, and you can use this for machine code programs and any extra screens or pictures you want to store, or even for extra-long BASIC programs.

PEEK and POKE

To get the most out of your computer you have to know what lies where in memory, and therefore what you can alter safely and what you cannot. Until now you may only have used memory implicitly, by programming and so on, though you have probably tampered with many locations directly, using POKE, as well as looking at locations using PEEK, such as sprites and sound etc.

If you are planning to use specific parts of memory then it is important to have an idea of what the memory looks like. In some respects you will already have a clear idea of certain parts of memory. You know how the sprites and the screen memory are organised, both for high and low resolution, and you will probably have some idea of the positioning of the BASIC program.

However, it is important to have a very clear idea not only of how you see memory, but how the the 6510 sees it too. In addition, matters are further complicated because the VIC II, the SID and the 6510 main processor all have different views of memory. Obviously it is important that you also know what these views are. All of this is especially important on the Commodore 64 because you are likely to be spending so much of your time PEEKing, and more importantly, POKEing round memory, due to the lack of explicit graphics and sound commands.

MEMORY ADDRESSING

Starting with an overview, remember that your Commodore 64 actually has 84K of memory. Of this, 64K is the standard RAM and 20K is the ROM, parts of which are transferred in and out of RAM as they are needed. In fact, the processor cannot see the ROM as it can only address 64K with its 16 bit address bus. To get round this problem the extra parts of memory are swapped in and out of view as they are needed. Some parts, like the 4K of character sets in ROM, are usually only looked at by the VIC II chip which can see 16K at a time. If you want, you can force the VIC II to look at any part of memory you like (see later on in this chapter).

To understand the way the 6510 addressing works (and this is the most important method, since it is the way that PEEK and POKE work) it can help to look at memory as a series of pages. This means that any address, which is two bytes long, can have one byte for addressing the page number and the other for addressing the location within that page. A decimal address of 1024, for example, will be address 0 in page 4 — the usual start.

STANDARD MEMORY MAP

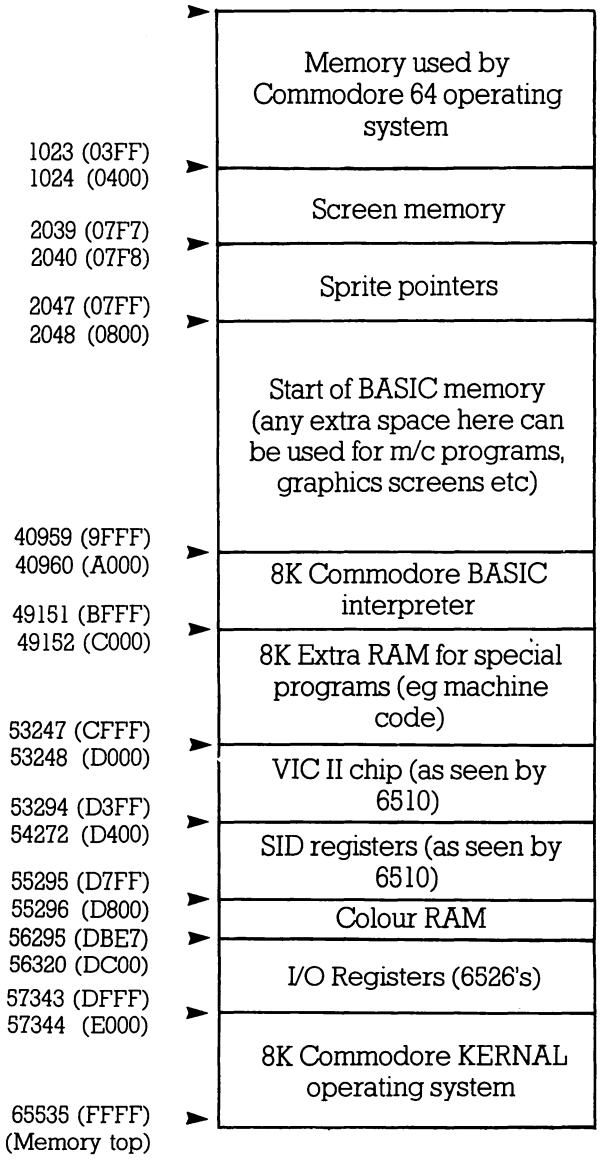
Moving on to practicalities, the simplest possible memory map for the Commodore 64 — that which is set up when you switch on and start programming in BASIC — is shown on page 107. (Numbers in brackets are hexadecimal).

Altering the memory map

Although this is the way memory appears to the 6510 when you switch on, you can change the order and the purpose of most areas yourself. You can even transfer the BASIC interpreter and the Kernal out of memory, giving yourself a bare machine with just 4K used for the I/O information. Although this leaves you with 60K free to use as you wish, you will have to perform any input or output yourself, and write the routines to do it. In addition, all your programs will obviously have to be in machine code, since there is no way of running BASIC, let alone of programming in it.

However, the fact that you can alter the pointers to parts of memory is very useful, and allows you features such as the ability to have two different BASIC programs in memory at the same time — even if their line numbers are identical.

In the original Commodore 64 *User Manual* there is virtually no information about how memory is laid out, with only the simple screen memory and colour RAM detailed,



Commodore 64 basic memory map

though of course the SID and VIC II base addresses are given. Much more detailed information is provided (as it ought to be) in the *Programmer's Reference Guide*, and if you want to know the contents of any important address then this would be well worth buying (if you do not already have a copy).

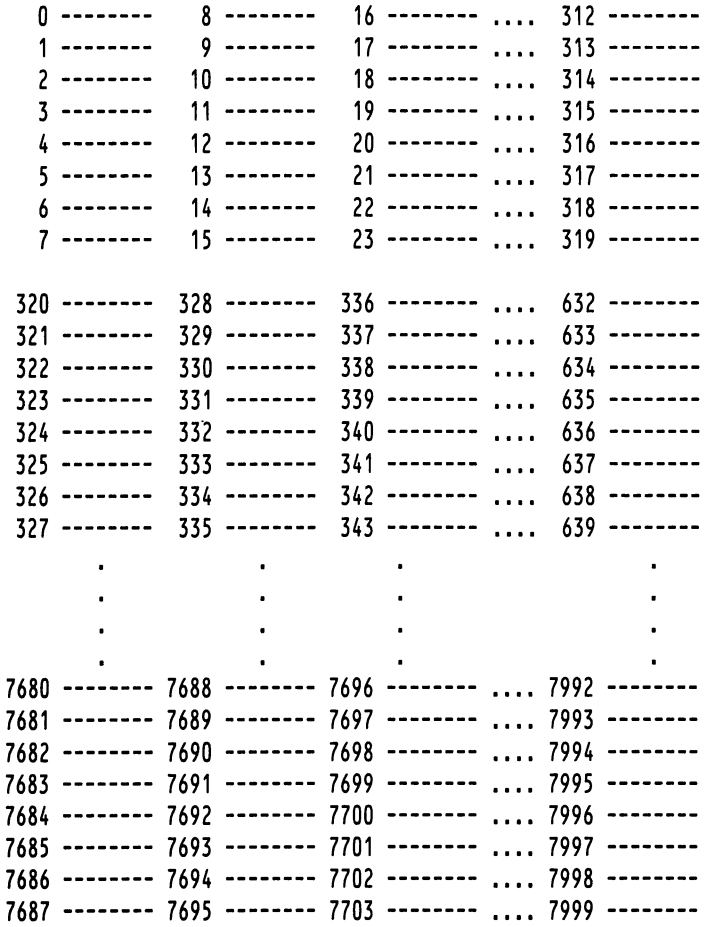
Once you know where things are, it can be interesting to have a look round memory using the PEEK statement. You can do no harm PEEKing around inside memory and examining the contents; it is only POKEing which is dangerous.

HIGH RESOLUTION SCREEN MEMORY LAYOUT

Most of the Commodore 64's memory is laid out with visible logic behind it, but some parts are not so easy to understand. The most obvious example of this is the layout of a high resolution screen (described in Chapter 7), which is organized as if the screen were still made up of characters. This layout makes the examination of 8x8 blocks of memory easy, and helps with the colouring, which is also arranged in 8×8 blocks, but can make looking across single pixel lines (either horizontally or vertically), quite difficult. Schematically, the bytes in a high resolution screen are laid out as shown overleaf, where each dash represents one screen pixel. The numbers are those for the position of the byte in high resolution memory (*NB* do not forget to add the base address to these numbers).

You can of course use both high and low resolution screen memory directly. Anything you POKE in will be passed on by the VIC II chip to the screen. Manipulation of high resolution memory is made quite complicated by the way it is arranged, and the expressions that you will need to isolate single lines, characters or points should be carefully thought out before they are applied.

You are more likely to be using this memory from machine code than from BASIC, as BASIC alterations to the 64,000 pixel positions are notoriously slow. You will also be manipulating the high resolution screen if you are employing the compaction and decompaction routines mentioned in Chapter 7, and in any other application where you are storing graphics explicitly and putting them onto the screen yourself.



High resolution screen memory map

COMMODORE 64 MEMORY

It is now worth having a closer look at the memory map, so that the individual problem areas can be seen. Starting at the bottom of memory, the first 1K is taken up with the Commodore's system workspace.

System workspace

The Commodore 64's operating system needs quite a lot of temporary work space, and so the first 1K of memory is reserved for this purpose. This area also stores the system variables, which are things like the start of BASIC pointer (addresses 43 and 44), the highest address used by BASIC (addresses 55 and 56), the number of characters in the

keyboard buffer (address 198) and many more. A full list of these is given in the *Programmer's Reference Guide*. You can PEEK the system variables to find out the status of almost anything in the Commodore 64, from the current screen colour RAM location (addresses 243 and 244) to the current size of the keyboard buffer (address 649).

System variables

Since these variables are in RAM you can also alter them by POKEing them with any value you want, although there are not many that you will want to change in the ordinary course of events. It is extremely dangerous to POKE the system variables, as even the smallest mistake may cause the whole machine to crash, and not just the BASIC interpreter. In many cases, the only way to get back to a working computer is by switching off and back on again, which resets all the system variables to their start positions.

In fact it can do no harm to the computer to POKE the system variables with anything you like, since the worst that can happen is that you cannot use the machine until it is reset by switching off and on again.

Nevertheless, if you have an important program running (or even resident in memory), or if you have anything that is not saved, like a machine code routine, then it is inadvisable to choose that moment to start POKEing the system variables.

The system variables give you a great deal of power, and used wisely can enhance your programming no end. One of the more obvious ways of making use of the system variables is in allowing yourself a different area for BASIC programs from that which currently exists (see the section on BASIC to find out how to do this).

To discover more about the meanings and normal contents of the system variables, look at the list given in the *Programmer's Reference Guide*. Remember that although they are called variables, they cannot be accessed like BASIC variables, but only by going straight to the memory location where they are stored, using PEEK and POKE. If your program changes any system variables at all, then look at these closely if you start getting peculiar errors.

After the system variables, the rest of the memory is organized according to the state of the system variables (or variables in other chips) which point to the location and extent of particular portions of memory.

Screen memory

Screen memory normally starts at location 1024 and is 1000 bytes long. This area can be moved to start at the beginning of any 4th page you like (ie at the beginning of any 1K block), but since this is done within the VIC II chip rather than in the system variables the explanation will be left until we are further up in memory. Unless you want to do something clever then it is generally a good idea to leave the low resolution screen memory at this location — where it is easy to remember where it is.

The position of screen memory is not likely to cause you any problems, though obviously obliterating part of it with something else in memory, like the BASIC program, will cause an odd display to appear on your screen. The VIC II chip interprets the bytes in the screen memory area and puts their corresponding characters onto the screen from the character matrix. Converting BASIC to screen memory will, at the very least, look peculiar.

Sprite pointers

Immediately above the screen memory there are eight bytes (2040–2047) used to store the sprite pointers. Your sprites can also be stored anywhere you like in the current 16K VIC window (see the section on the VIC II chip for details of this). Each of your eight possible sprites must have a pointer here to the start of the sprite in the sprite table. If your sprite table starts at, for example, page 12 (location 12288), the number in sprite pointer 0 should be 192, since 12288 is 192×64.

If your sprite table was consecutive then sprite 1 would follow at 193 (193×64=12352), leaving 64 bytes between the sprite start positions, for the sprite data.

One important thing to remember about the sprite pointers is that if you move the screen you should move the sprite pointers at the same time, as the VIC II will expect to find them at the end of the screen memory. Failing to do this can give apparently inexplicable errors, caused by the sprite pointers being lost, or the VIC II assuming that whatever is after the screen are the sprite pointers — even if this is not true.

Basic program and variables

If you have just switched on your Commodore 64 then the next thing you will find, above the sprite pointers, is the space for BASIC programs — or, if you have already typed

one in, your BASIC program. This is a large chunk of memory which you can use pretty much as you wish, the next major obstacle, the BASIC interpreter, starting at location 40960. This space, as you will see when you switch on, is 38911 bytes long (location 2048 itself being a zero, as was described before), and can be used for programs, machine code or stored graphics.

BASIC pointers

You can easily find out where your BASIC program lies by PEEKing the system variable at locations 43 and 44, which gives the start of BASIC, and you can find out how long it is by PEEKing the start of BASIC variables pointer, which is at locations 45 and 46. However, you really ought to include the length of the variables and arrays with the length of your BASIC program. The pointer you need to examine is the start of dynamic strings pointer, at locations 51 and 52, but unfortunately even this does not give you a precise answer.

The reason for this is that the variable storage in a BASIC program is dynamic — that is to say it alters when you run the program. This means that to find the length of a program you really need to look at the pointers *after* the program has run.

Space left after using BASIC

This will give you an indication of how much free space you are left with. You will need this space if you want to save whole screens, or if you want to get compacted screens and machine code tucked away in memory. If you try the exercise of looking at the pointers with both a long and a short program you can see the different amounts of memory that you might expect to have free.

You can place a limitation on the amount of space which BASIC can physically access by moving the pointer stored at locations 55 and 56, the last address usable by BASIC. Doing this will give you protected space above the top of BASIC programs and below the BASIC interpreter, but will also give you correspondingly less space for your programs. The contents of 55 and 56 are normally 0 and 160, giving 40960 as the top of usable BASIC. This pointer can be lowered as far as you wish, though if it is too low you will not have sufficient space for BASIC.

Notice that each of these system variables is two bytes long because you need two bytes to store a complete 16 bit memory address. To get the address in decimal from these

two numbers you can take the contents of the second of the two bytes, multiply it by 256 and add the contents of the first, since the bytes are stored as low, followed by high.

When you are looking at the BASIC pointers, bear in mind that some programs may not take up much BASIC space, but they can take up a lot of variable space. For example, if you set up a two-dimensional string array then you can find that the variable space needed to store it is enormous. This can affect any calculations you may have made about the amount of memory that you can clear by lowering the top of BASIC pointer and still be safe.

How structure affects space

Another factor which can affect the amount of memory you have spare is the structure of the program. If you use recursive subroutines then this can use a lot of stack space, and you may not have taken this into account at all. Some of the more efficient sorting routines use both a lot of memory space for the variables and a lot of stack space, if they are recursive. The result of this can be that the BASIC program may look very small but in fact could cause large amounts of memory to be used when run, outside of the area used by the BASIC program.

A short routine which PEEKs the relevant system variables and prints out their values can come in handy here, so that you can find out where to store information after you know how much memory is going to be used by the program.

```
10 INPUT N
20 LET X=PEEK(N):LET Y=PEEK(N+1)
30 LET V=(Y*256)+N
40 PRINT "SYSTEM VARIABLE AT "; N ; " IS"; V
50 GOTO 10
```

If your machine code is relocatable you can then move this into position once you know how much free space there is.

In practice, if you know what your program and any machine code routines are going to do, you can estimate what sort of quantity of memory is going to be used up by the BASIC, and how much space you will have to leave for the machine code. In most applications in any case, your machine code routines are not going to be put immediately above BASIC, but in the 4K of RAM which lies above the BASIC interpreter and below the VIC II.

Writing two BASIC programs at once

One trick which you can do once you know about the pointers of the BASIC program is to write two separate programs in BASIC at the same time. As you remember, the start of BASIC pointer in locations 43 and 44 is currently pointing to location 1 in page 8 (address 2049, since address 2048 is required to be zero by the operating system). If you raise this pointer (lowering it can be disastrous!) then any BASIC program you have in memory will be protected and can only be altered by a direct POKE. You can now write a new program at the new location pointed to by 43 and 44, and it will not affect the old one.

However, if you are going to do this then it is advisable to move the variable and array pointers up at the same time, otherwise you could find yourself with the variables and arrays underneath the BASIC program, which is not considered healthy. You can then reset the pointers, and the upper program will be protected and the lower one can be manipulated again. This technique is frequently used in games, and is very powerful. It is also very easy to lose one or both of the programs doing this, since one slip of a pointer can be fatal.

How BASIC programs are stored

If you are going to be conversant with the inside of your computer then it pays to be aware of the way that BASIC programs are stored. Each line is stored as a series of bytes, and each keyword reduced to a single number.

The program statements or instructions are linked together by what are known as link addresses: preceding each program line there is a link address saying where the start of the next program line is stored. The program is run according to these links, and if you want to manipulate the program then you will need the link addresses.

Renumber routine

Probably the commonest application you may have seen which uses the link addresses is the renumber routine. This can look through the program examining the link addresses and changing the line numbers. At the same time it will be looking for the codes for keywords which use line numbers, like GOTO, GOSUB etc, and changing the following line numbers to suit the new version of the program being created. In practice, it is difficult to write an efficient renumbering routine, but the theory of it is simpler than you might have imagined.

Program protection

The link addresses of a program can also be used to give you a simple protection method for your BASIC programs. Just type in the following statement:

PRINT PEEK(2049),PEEK(2050)

This will produce two numbers: the link address. If the first program statement was a REM then the link address will give 19 and 8 as the two numbers. Note these down, and then type in the statement:

POKE 2049,0:POKE 2050,0

Now SAVE the program. You will find that as there is no link address there is effectively no program, and LIST and RUN will have no effect. After you have loaded the program again you can put the link address back in with the following statement:

POKE 2049,19:POKE 2050,8

Without the address (which you have kept secret), other people will find it difficult to break into your program. And this is a very simple measure. Imagine if you went through the program changing all of the link addresses. You could then add a machine code loader to POKE the numbers back in automatically.

BASIC interpreter

After the large space there is 8K reserved for the Commodore BASIC interpreter from locations 40960 (A000) to 49151 (BFFF). This can be transferred out of memory if you do not wish to use BASIC, though obviously control (once it has gone) is in your hands — or in those of your machine code. On the whole, unless you are going to write the most sophisticated software, you will probably be better off leaving the interpreter right where it is.

Extra RAM

From locations 49152 (C000) to 53247 (CFFF) there is a 4K space. This is an ideal place to put machine code routines, as long as they are less than 4K in size. If they are put here there is no way that BASIC can touch them as the interpreter comes between the two.

This area means that your machine code does not necessarily have to be relocatable, though it will be more useful if it is. Relocatable machine code is written so that it will work in any position in memory. The idea behind it is

that instead of putting absolute positions into the machine code programs, you make everything relative to the first position. If this first position is changed then everything else changes with it.

Note that it is only possible to do this with short machine code routines. With longer routines the jumps can no longer all be made relative — some must be absolute (see Chapter 12 for further details).

VIC II

Above this blank area of RAM there is the space for the VIC II. It is important to realise that the VIC II is itself pretty sophisticated, and that it looks at a block of memory in the same way as the 6510. The confusing problem is that it looks at the memory in one way and the 6510 looks at it in another, and their views only overlap in places.

VIC II windows

The first major difference is that the VIC II only sees 16K of memory, so to give it access to the whole 64K of memory you have to give it a window on the Commodore 64. This window can be in one of four positions, and is initially set to 0, matching the bottom 16K of the computer.

The least significant two bits in location 56578 control which 16K of memory the VIC II sees, and any block can be chosen, as follows:

Bits	Window start	Window finish
11	0 ($0000)	16383 ($3FFF)
10	16384 ($4000)	32767 ($7FFF)
01	32768 ($8000)	49151 ($BFFF)
00	49152 ($C000)	65535 ($FFFF)

All numbers prefixed by $ are hexadecimal

However, in two out of the four blocks the VIC II cannot see the 4K ROM image of the character set. When it is set to window number 0 or 2 (locations 0–16383 or 32768–49151), the VIC II sees the character sets. In the other two positions the image must be picked up explicitly from RAM.

VIC II registers

From the 6510's point of view the VIC II registers are the important part, and it is these which are controllable from BASIC or machine code at the VIC register locations (53248–53294). You will have used these already, though some may not have been examined. One which is quite

important is the pointer to the offset for the screen memory, within the current VIC window. This is normally, as you have seen, set to 1K (1024), but can be set to any of the start addresses of a 1K block (ie 1024, 2048, 3072 etc). The register which holds this is the top four bits of location 53272. Taken as half a byte, the number here simply points to the K at which screen memory starts — normally the first four bits are set to 0001 accordingly.

Most of the other registers have more frequent use, as they are used in controlling anything to do with the graphics — as you will have already seen in Chapters 7 and 8.

SID registers

These registers are located immediately above the VIC II (from location 54272), and, because the SID is yet another chip, are treated in much the same way as the VIC's. The SID does not affect memory in the same way as the VIC II, but the registers are used and manipulated in a similar fashion. You should not encounter too many problems with the placing of these, though of course using sound is one of the most complex and difficult areas of programming on your computer (see Chapter 9).

Colour RAM

Above the SID is an area of memory which you will already know very well: the colour RAM (locations 55296 (D800) to 56319 (DBFF)). The colour RAM, as you will remember from Chapter 7 and earlier, defines the colour of each 8×8 square on the screen.

These locations can be POKEd with new values if you want to change specific squares without altering the whole screen, but you will probably use the character commands mostly from the keyboard, as these can be used comprehensively rather than on single characters alone. However, within loops you can effectively change the colours of a single horizontal or vertical line, or any regular-shaped area, by POKEing this area of memory.

I/O registers

The 6526 CIA chips, which handle almost all of the input and output to the 64, lie above the colour RAM from locations 56320 (DC00) to 57343 (DFFF). You will have seen how these are used in Chapters 5 and 6 which deal specifically with input to, and output from the Commodore 64. The registers in the chips are very powerful, and if you do not wish to use the Kernal then you can push everything out of memory except

these chips, for without them the computer is powerless. Input and output are crucial — treat them with care, or leave well alone.

The Kernal

Right at the top of memory there is an 8K machine code program, or rather suite of programs, called the Kernal. This is the operating system of the Commodore 64 and handles the way the machine works. Without the Kernal you would not have anything, since the interpreter calls many Kernal routines. The Kernal also sets up all the startup and input/output procedures, as well as the way in which the VIC, SID and CIA chips work. Altogether, the Kernal is pretty busy, as you might guess.

It is inadvisable to try to modify any of the Kernal routines, but most can be accessed by the user, and in fact from machine code they are essential. Typical of those you might use are routines like ACPTR at location 65445, which inputs a byte from the serial port.

Another is SCNKEY at location 65439, which scans the keyboard. To get the best from the Kernal you should leave it alone until you start programming in machine code (see Chapter 12 for details of this).

STORING MACHINE CODE

If you are really stuck for space then you can push short machine code routines into places where they should not really go. There are all sorts of odd spaces left unfilled in the Commodore 64 and some of these can be used by you if your machine code is short enough. For example, among the system variables there is a gap from locations 679 to 767, and if you have a routine shorter than 88 bytes then there is no reason why you should not use this area.

You will soon discover one of the golden rules of computing: no matter how much memory you have, you will be able to use it all up before you have done everything you wanted to. Once this rule has been discovered, people will find incredibly obscure ways of using odd bytes here and there, and some of the best machine code programs do just this. People get very devious when it comes to making the best possible use of a limited resource.

The ingenuity of some ideas has to be seen to be believed. There was an apocryphal case when a mathematical program had to do huge quantities of calculations, but there was not enough space for it to do them. The

programmer solved the problem by switching off the screen for a while, redirecting the workspace to the screen memory at locations 1024 to 2023. When the calculation was finished the pointers were reset, and then the results printed out.

COMMON MEMORY PROBLEMS

Some of the more common memory problems are worth detailing, if only because they are easy mistakes to make. One that happens all the time is that of loading in several machine code files, and getting them overlapped by mistake. When you load in machine code you do not usually know where it is going to go (unless you are unusually efficient), and so it is a good idea if you can keep a note with the cassette or disk of the start location and also of the extent of the code.

The difference between loading in 100 bytes and 4K is obviously going to affect other programs. If you can be sure always to make this note of the address and extent, then the problem will never affect you, but even the most organized of us sometimes forgets the location, and loads in two machine code files, one of which wipes out part of the other. This error is not always discovered until something important has been written, and subsequently lost by running the erroneous machine code.

Machine code files which start in the same place, or ones which would overlap when loaded, need not cause problems if you know the size and start details. They can simply be loaded in and then moved as necessary, before loading in the next file. To do this correctly, of course, the programs do need to be relocatable.

An even simpler problem is that of loading the machine code into the wrong place. If you do this then you are almost certain to get problems. If this has happened it is unlikely that you will have got away with putting it in a safe place — it is more likely to arrive somewhere critically important. In any case the effect is disastrous, and the computer will usually crash because of it. The likely cause is that you have loaded in a machine code file to the wrong address.

Even if you have all of your machine code in the right place, there is still plenty of room for error. If you should accidentally use the wrong number in the SYS statement calling the machine code, then the Commodore 64 will try and run a machine code routine that either does not exist or started before or after the location called. If you get a

problem with a machine code program check that you are actually calling it, because otherwise you could end up wondering why it does not work properly, when in fact it was the call statement which was wrong.

POKEing almost anything into memory can cause confusion too. This is not to say that you should not do it, but that when you do, you should be particularly careful. This is especially important if you are POKEing into screen memory directly (which you must do in high resolution), or into the system variables.

The final problem with the Commodore 64's memory is that of running out of it during a program run. This is obviously not a satisfactory state of affairs, but it is important to find out exactly why you ran out of memory before rushing in and just giving yourself more room by moving a few pointers.

The most likely causes are that you have tried to declare an array or arrays larger than the space available, or that a recursive subroutine has got stuck inside itself and has continued until the stack has run out of space into which it can expand. In either case it is just a question of modifying the program to use less space. Unfortunately there is no easy way round the problem.

11

STRUCTURE

Most of this book has been devoted to the problems which crop up while programming. These and the sections on identifying and curing bugs have all made the assumption that the program has already been written. As it happens, many of these problems could have been avoided if the program had been designed and written more carefully. And many others could have been more easily identified and cured if the program had been written with the possible problems in mind. It has always struck me as an extraordinary fact that something over ninety per cent of all commercial programming time is spent debugging or modifying programs. This seems to be an enormous waste of resources.

PROGRAM DESIGN

The only stumbling block you have to get over, before you can design and write programs in a better way, is your attitude. People's attitude to program design has always been negative. Mention subjects such as structure, documentation or design (let alone flowcharts and block diagrams!), and most people will shy away. The majority of programmers still believe that although design ideas may work in theory, sitting down and writing structured and documented programs will, in practice, require a lot of effort.

REDUCING DEBUGGING TIME

It may be true that it takes some effort to write a structured program, but the benefits you will reap far outweigh the extra time and effort spent designing. Any programmer will tell you that you spend more time debugging, correcting, modifying and altering a program, than you ever do writing it in the first place.

It therefore stands to reason that if you can cut down on the debugging and correcting time, and make a program easier to alter or modify, then you will save far more time than it took you to write it at the outset.

Once you are in the habit of writing organized programs you will find that it takes no longer to do so than it used to

take you to write a disorganized one, without any of the advantages a good design brings.

PROGRAM FLOW

So, what makes a good design? What is a structured program really like? To answer these questions it is necessary to grasp one of the basic facts of programming: a clear program flow is the key to it all. When a program runs it flows, and this happens basically from the top of the program to the bottom. On its way down the flow may jump around, go round loops and branch off to different parts of the program, but the program is flowing all the time.

If you imagine the flow path as being a piece of string, with knots representing bugs, then a structured program is a piece of string which you can follow from one end to the other without getting confused. An unstructured program looks like a plate of spaghetti. Continuing the analogy, you are far more likely to get a knot in the spaghetti than you are in the simple line. But if a knot does occur, you are much more likely to be able to untangle it, if you are not the producer of spaghetti programs.

The analogy works for modifications too. If you want to cut the spaghetti and add a bit into it, without losing the flow path, it is going to be very difficult. If you want to do this with the straight string it should be (relatively) simple.

PROBLEM SOLVING

So much for the theory. What you need to know now is how to structure a program to avoid spaghetti. As it turns out, the only way to get a structured program from the outset is to start before the programming stage. Look at the problem you are going to solve (for all programming is basically problem solving), and work out how it could be split up into smaller tasks. For example, if you were to draw a city at night it could be split up as follows:

1. Colour screen/border black and switch to high resolution
2. Draw city outline
3. Draw a few tower blocks
4. Draw lighted windows in tower blocks
5. Add some stars randomly above the outline
6. Add the moon

Splitting the problem into sections

Each of these can now be looked at as a separate program. In fact if each of the six parts were to be programmed separately this would be a very good way to start. Without knowing anything much about structure you will find that if you do this, the result is a structured program, the flow of which can be easily followed.

Top-down analysis

This is actually something which academics call top-down analysis, where you start at the top with the problem, and work your way down to the solution (the program). You do not need academic names to practice the art of structure, as you can see.

Testing of small units

When you are programming each section, you can also test it, to see if it works as you want it to, without ever needing to know if the rest of the program works. Once each part is finished it can be saved to tape, and later on merged into the final program, which is simply the six sub-sections linked together. You need only do this after all the separate parts work.

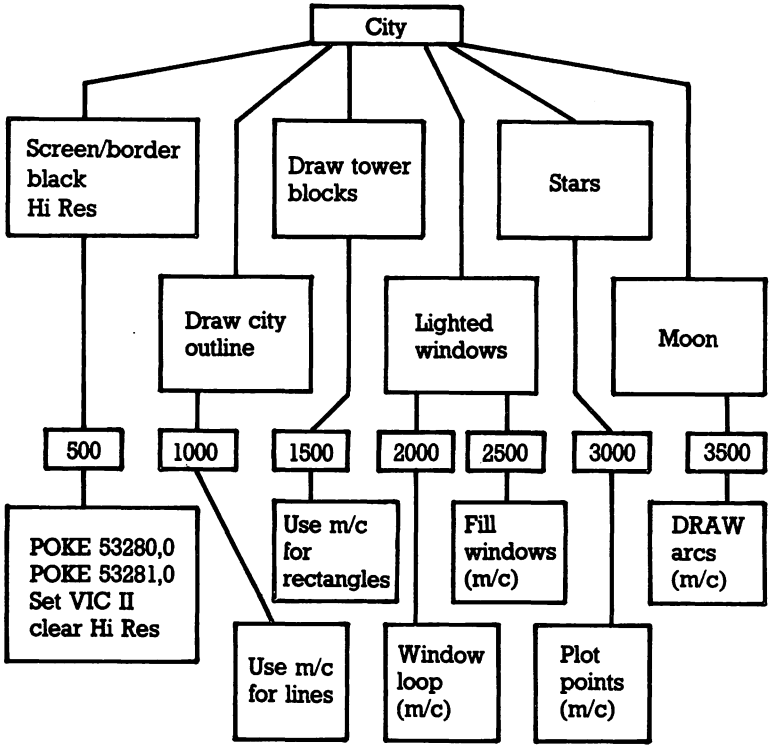
Block structuring

In the process of programming any of the individual parts of the program you might find a place where you could logically split the task still further. One example of this in the city program above is Section 4, where the logical approach would be to have one routine which draws windows and another which colours them in. This again follows the ideas of block structuring, where the theory is that you keep reducing the size of the task until it is easily programmable.

Even if you ignore the rest of this chapter, remember this advice when writing programs: *any problem which is too big to be programmed in a few lines should be split into several smaller problems.*

Incidentally, the splitting up of the problem may be shown formally as in the figure at the top of page 124.

Underneath the sections already described it can be helpful to jot down ideas before going on to write the program, including the lines you want to use for each section and the method of solving the problems. In this case you can see in advance that the drawing is going to be difficult and the idea of calling machine code from BASIC has been noted down.



Although, in practice, it seems unlikely (out of laziness) that you will start drawing a plan like this, you can see that the logic is much clearer than it was when the idea was conceived. Anyway, it is a good idea to know what you might do, if you were being painstaking. Most professional or really important programs are written using this method.

Merging sections together

When you merge the separate parts together be careful not to make any obvious mistakes, like using the same line numbers for two or more of the modules. This is avoided by the method used above: where you put the start line number of each section is shown. If, after merging, the program does not now do what you wanted, you will know that the problem is in the way they interact, since each one works on its own. This means you only have to look at the effect of each routine on the others, and you should find the error quite easily.

If you had started with an idea and just dived into it, only to find that it did not work, then you would have to look at the whole program. The advantage of this method is that you know that all of the separate parts work already, since they were written as self-contained programs.

This is the practical and useful side of structured programming: there are a great number of theories about it too, but since these mostly revolve around languages which avoid the need for the GOTO statement, or which have other constructs built in (like CASE, WHILE and REPEAT) which help keep the flow ordered and smooth, there seems little point in saying much about them here.

STRUCTURE PROBLEMS IN COMMODORE BASIC

Commodore 64 BASIC, as a language, is really not conducive to this kind of theoretical structuring. It would be nice if it had some of the constructs mentioned above as they all make a programmer's life much easier.

Obviously you can implement the simpler ones like REPEAT and WHILE yourself (see Chapters 3 and 4), but even so you are doing this with basically unstructured statements like IF and GOTO.

Simon's BASIC

There is an answer to your problems with Commodore BASIC and that is Commodore's Simon's BASIC. This is a cartridge which enhances the language enormously, not only giving you many constructs used in structured languages but also extra facilities like graphics commands. There is of course a sting in the tail, in that it is quite expensive and programs written using it will not work without it. This said, it adds a lot to the Commodore 64, and I would not program for my own use without it.

GOTO

Going back to the usual Commodore 64, the idea behind avoiding GOTO is worth mentioning briefly, as it throws some light on the reasons why so many programs end up with a spaghetti-like flow. If after every IF statement you could insert all the lines that logically followed it, without having to use GOTOs, then life would be easier, and the program would be much easier to read.

Some languages (notably Pascal and Algol) include the facility to have as much program as you like after the word THEN. When the program reaches an END statement then it knows it has reached the end of the THEN part of the program, and continues. In this way many parts of the program appear where you would expect them to — the opposite of what often happens in BASIC.

In BASIC you have to divert the program to one place if a condition is true, to another if it is not, and you have to have a separate diversion to avoid the two bits running into one another. By comparison, it really is very nice to program in Pascal, though I have yet to see a version for micros which is as good as ones available on larger computers.

Since these are academic points, they are not really worth dwelling on. More to the point is to look at other ways you can make use of the facilities the Commodore 64 offers. One definitely under-utilised by almost all programmers is the REM statement.

REM STATEMENTS

If you decide that you are going to program in small modules then there is nothing more helpful than a REM at the beginning to explain the purpose of that particular module. This means that when they are all strung together you get a picture of the whole program, and how it was programmed. It is also worth using a REM statement anywhere in the program where what you are doing is not obvious — you may be using a trick, or just something complicated, but if there is a REM nearby, it will save you wondering in six months time what it was you were trying to do six months ago. This is particularly applicable to the Commodore 64 since many statements are going to be POKEs into the VIC II and the SID, and the meaning of these is rarely obvious — even to the knowledgable eye.

Remember, all of these ideas about structure and documentation are there for your benefit, and not because some academic says they should be. They are all ideas that are aiming to help you have an easier time of programming. If you read through a program which was built out of modules, and which is liberally sprinkled with REMs you will probably find that you can see what it is up to, almost straight away. This is rarely the case with the average program you glance through.

Since, if you are trying to hunt out the errors in a program, a readable program is half the battle won, the REM statements are worth their weight in gold. Admittedly, if you are very short of space then you can save some by leaving out REM statements, but the prospect having to debug a program of that size, without the aid of REM statements, would be daunting indeed. There would be a strong argument for looking for another solution, if this was the case — like machine code, for example.

One of the big advantages of building up your program in modules (or structuring it, if you like) is that of easy modification. To modify a modular program you can drop one of the existing modules and slot in another, or you can add new ones, or lose old ones, all with the minimum of fuss.

PROGRAM SHELL

There is one more practical idea which you can incorporate into your programs very simply, and that is the program shell.

The example below is actually a programmed version of the splitting up process that you saw for the city by night program. This time you could start the program as follows:

```
100 REM CITY BY NIGHT PROGRAM
110 REM SCREEN/BORDER BLACK AND HI-RES
115 GOSUB 500
120 REM DRAW CITY OUTLINE
125 GOSUB 1000
130 REM DRAW TOWER BLOCKS
135 GOSUB 1500
140 REM DRAW LIGHTED WINDOWS
145 GOSUB 2000
150 GOSUB 2500
155 REM ADD RANDOM STARS
160 GOSUB 3000
165 REM ADD THE MOON
170 GOSUB 3500
190 GOTO 190
200 END
```

Using subroutines

Once you have done this you can program each part as a subroutine, and end each with a RETURN statement. To the reader of the program it is quite obvious what is going on, and the flow here is a perfectly smooth line. There may be less easy constructs and flows in the individual subroutines (including jumping to and from machine code), but the main program is very easy to read. At first the user does not even have to know how the subroutines work, but can just accept them as programmed tasks.

If one subroutine goes wrong then it is only that one which needs examining. This idea of the program shell is almost universally used now by professional programmers, for the simple reason that it works so successfully.

Ending with an infinite loop

Notice that the program above effectively ended with an infinite loop at line 190. This is one of the best ways of ending any program that has graphics displayed from it, because if you just use a simple END command, the computer returns to immediate mode, with READY, as soon as the drawing is finished, probably obliterating some of the things you have drawn. In fact this program does not need an END after the GOTO in 190, but it is much neater if it has one. Without the GOTO or END the program would run straight into the first subroutine and when it found the RETURN the program would crash, despite there being nothing seriously the matter with it. Apart from anything else, the END tells the reader of the program that line 200 is where the shell finishes.

Other things to note about the program are that the subroutines have been given 500 lines each, so that there is plenty of space for them (more than enough!), and the shell itself has four blank lines for every one used. This means that if you want to add in any other routines or ideas then it will be easy to do so. There is another reason for leaving plenty of blank lines, and that is so that you can add in temporary PRINT statements if things are not going according to plan. These extra statements allow you to examine the variables, and the way in which they change.

The shell also allows you to edit out whole sections of the program very simply. To run just one of the six separate parts of the program you only need to put a REM into the call for each of the other five routines, and they will have been edited out — until you take the REMs back out again. In this way you can have the whole shell present, but only a small part of the program being used.

A large space has also been left before the very first line. This is so that there is plenty of room for you to define any functions you want to, at the top of the program, and so that any variables can be initialised where they are easy to see and easy to change at a later date. If your functions and variables are always described first near the top of the program, then you will not have any problems in working out what each one does, but if they are scattered apparently randomly around the program, you may have a lot more trouble chasing them up and finding out what they all do.

Variable names

When you are choosing variable names it helps to be as explicit as possible, since a name like 'X' is not nearly as useful to the reader of the program (probably you) as something like 'MOON'. Again, these ideas are there for your benefit, not for academics, and whether or not you use them is up to you. However, if you come back to spaghetti land after structuring things for a while you may wonder how you ever managed to program anything at all.

12
MACHINE CODE

WHAT IS MACHINE CODE?

Machine code is just another computer language, like BASIC. It is the language which your Commodore uses to run everything, and consists of many very simple instructions, which can be acted on very quickly. It is far more laborious and difficult to write than BASIC since, in its pure form, a machine code program is simply a long list of numbers between 0 and 255 — ie numbers which fit into one byte each. Each of these either represents an instruction or a piece of data for an instruction.

Because the Commodore only understands machine code, every line of your BASIC program has to be translated or interpreted into machine code before it can be run. For this reason programs written directly in machine code, rather than BASIC, are almost bound to run much faster. It is interesting to note that the largest machine code programs you have regularly used are probably Commodore BASIC itself and the Kernal (the operating system).

The 6510

To understand how machine code works it is a good idea to have a look at the nature of the machine for which you are going to write code. The heart of the Commodore is the 6510 microprocessor, and it is in the language of this that you want to write.

6510 assembler is not as difficult a language to learn as you might imagine, though obviously it is both more laborious and complex than BASIC. Fortunately if you have used a BBC Micro, Electron or Atari then you will be well on the path, since the instruction set for the Commodore's 6510 is identical to that of the 6502 and 6502A processors. The language you need to learn is actually 6502 assembler.

The registers

Inside the 6510 there are a number of permanent hardware variables called registers, including the principle one, A, which is used as the accumulator — rather like the memory on a simple pocket calculator.

Each of these registers can hold one byte, and the different instructions you use all relate to manipulating information with or within these registers. For example, a byte can be loaded from one place in memory, and another can then be added to it. The result can then be stored away in a different place in memory. These three actions are the equivalent of an instruction as simple as LET A=B+C in BASIC. And even then you could only have added two numbers with a combined result of less than 256! The main advantage is that you could do several hundred machine code additions in the time it took you to do one in BASIC.

The Flag register (Processor Status register)

There is also a special register called the Process Status register which has a different use for each bit. These bits of the P register are called the flags, and signify the results of previous operations. For example, if the last operation caused a carry to be made (eg during subtraction) then the carry flag is set. In addition there are flags which detect a zero result, a result which was negative, and a result which caused an overflow.

The Stack Pointer and Program Counter

As well as the registers there are several other features of 6502 machine language which make a programmer's life much easier. One of these is the stack, which is like an array in memory which can be used as a temporary storage space — you can push numbers onto it, which can be subsequently pulled off again. There is a special register kept free so that you can find out where the last piece of information was pushed on, and this is called the Stack Pointer, or SP.

Another special register is the Program Counter, or PC, which keeps track of the next machine code instruction to be executed. Branches in the program are effected by changing the value of the PC. Of the six main registers only the PC is a sixteen bit register, while all the rest are eight bit. The reason for this is obvious — the PC needs to be able to address all of memory (65536 locations) while the others only manipulate numbers (the SP only addresses a 256 byte area of memory).

WHERE TO STORE MACHINE CODE

Machine code programs can be stored almost anywhere you like, but you should be aware of the memory map before placing machine code in any particular position. The best place to store machine code is high in memory, in the 4K

space from 48K to 52K ($C000 to $CFFF). If you use this area then instructions like NEW will not wipe out your machine code when the BASIC area is cleared.

INTERPRETERS AND COMPILERS

When people talk about machine code they frequently get confused between interpreted, compiled and assembled machine code. When writing machine code yourself you are only likely to be using the assembled sort, since interpreters and compilers have different functions. An interpreter is the device which enables you to program in BASIC and then have the program interpreted into machine code and run, line by line. This is what happens to all BASIC programs on the Commodore 64. A compiler on the other hand performs the same sort of function, but operates on a whole program at once, rather than translating it line by line. All that these two devices do is enable you to run your BASIC programs at all, though compiled machine code will run faster than interpreted. By using these aids you are not programming in machine code, but in BASIC.

The best troubleshooting tip for users of compilers and interpreters is to look back at the earlier part of this book, where BASIC troubleshooting is covered. The point about compilers is that your BASIC program should work without being compiled, and so the troubleshooting techniques in the rest of the book apply. Once the program is free from bugs it can then be speeded up by whatever method you can find.

If you are not using interpreted or compiled machine code then there are two other methods you can use of actually programming in machine code. These are assembly language and hexadecimal programming.

PROGRAMMING IN HEXADECIMAL

In practice it is extremely unlikely that you will ever program your Commodore 64 in hexadecimal. Programming in hexadecimal is a two-fold process, and basically involves you providing the numbers that make up the language yourself. The only way to program in hexadecimal is to write in 6502 assembly language, and then convert it by hand into the relevant hexadecimal instructions — the numbers between 0 and 255, described earlier.

In certain instances this is not too taxing, but when it comes to counting the number of bytes to be jumped over, or

the value of the current program counter (PC), it can get rather complicated. Apart from anything else, it is very hard to debug. You do not usually have the same range of tools to help you as when you are trying to track down and remove any bugs in assembler. A pure machine code program gives few hints as to what is going on since it is literally just a list of numbers.

In fact, hardly any machine code programmers actually program in machine code; most use assembly language. This is then assembled to form the machine code itself. The only time when you might use straight machine code is when programming a computer for which there is no assembler — and even then you would probably find that the quickest route would be to write an assembler and then use it to write your other programs!

ASSEMBLERS AND ASSEMBLY LANGUAGE

Any assembler will translate a 6502 assembler program into the binary instructions (hexadecimal, in practice) which the 6510 understands. 6502 assembler is not as difficult a language to learn as you might imagine, though obviously it is both more laborious and complex than BASIC.

Mnemonics

Three letter codes, called mnemonics, are used to indicate the meaning of each instruction, and these are the main reason why 6502 assembler is more intelligible than hexadecimal. Here are some typical 6502 instructions, the meanings of which are fairly obvious:

LDA $0314	Load register A with the contents of hexadecimal location 0314
RTS	Return from subroutine to the place from which it was called
PHA	Push register A onto the stack
LSR A	Logical shift right on register A
DEX	Decrement the index register X
BEQ EX	Branch if the zero flag is set to label EX

As you can see they all bear some relation to the sort of instructions you might find in a very low level BASIC program.

Labels and comments

Another feature assembly language offers is that of allowing you to place labels (reference markers) in a program. The 6510's equivalent of the GOTO and IF statements can then

jump to these labels (as you saw in the BEQ EX example, above). Assembly also allows you to document programs, by placing a semi-colon after an instruction. Everything written after the semi-colon is ignored, in the same way that everything after the word REM is ignored in BASIC programs.

Programming tools

Despite the power of an assembler it is not in fact as much use on its own as you might imagine. To make the most of machine code you also need some tools. In principle it is the same as having a block of wood — with the right tools it can become a piece of furniture, without them you can end up with blunt fingernails and broken teeth.

A 6502 manual

Perhaps the first "tool" you need is a good guide to the 6502 instruction set. This does not need to be specific to the Commodore 64 (although this can help), but it must detail all of the instructions in 6502 assembly language, as well as explaining clearly the addressing modes, the registers and the assembly process. I still use a university textbook, which is long since out of date, but which does explain the language — you may well prefer to use one of the more recent specific Commodore 64 machine code books.

A combined editor/assembler

Next on your shopping list should be an editor/assembler. In practice you are unlikely to get an assembler without getting an editor too. This allows you to write machine code routines in 6502 assembler, on your screen, and to edit them as you would text on a word processor.

This is an important feature, as, since there are no explicit line numbers, new lines of assembly must be placed in the routine exactly where they are needed. In some of the Commodore assemblers and editors you are provided with line numbers, but these can be more of a hindrance than a help.

Some people I know actually resort to writing their machine code programs on a word processor, and then assembling the text file, rather than bothering with an inefficient editor. One of the better assemblers available is Commodore's own cartridge, though there are several alternatives which bear examination before purchase.

A good editor will allow you to use global search and replace within your program. This is a particularly useful

feature if you decide to change a variable name — finding all of the occurrences of a particular variable in a machine code program can be well nigh impossible.

Assembling a machine code program

When you have finished writing the program using the editor, you then assemble it. The best assemblers will do other things for you at this stage too, including error checking (ideally stopping at the line of assembly which was rejected), and syntax checking (making sure you have not misused the 6502 instruction set). The assembler will also place the assembled program into memory at the place you designate, and it can be saved from here to tape or disk.

In addition to an assembler you would do well to acquire a compatible machine code debugger, or test tool. Unfortunately these are scarce, but without one you have to confine the majority of your troubleshooting to checking and tracing before running the program. With a suitable test tool you can check on things while and after the program has run, with greater ease.

MACHINE CODE DEBUGGERS/TEST TOOLS

Trace facilities

When it comes to ease of debugging and general troubleshooting, a machine code test tool is very helpful. It will allow you to look closely at the way a program is running, and even, if necessary, to run it instruction by instruction. This single stepping is obviously going to be your last resort, since the looping of machine code instructions makes the total number of steps huge, in most cases. This error-trapping by tracing through a program, instruction by instruction, is only effectively used once the error has been trapped to within five or ten lines of code.

Examination commands

Other features which should be incorporated in a debugging tool are examination facilities. With these you can look at the current state of the registers or at specific locations in memory. You should be able to tell a lot about what is happening in the program from this, since the registers are your working variables and memory is where you will be storing results and other information.

Examining the contents of registers and memory (and the PC and SP) should show you how well the program is progressing. Careful use can show you precisely where a program is going wrong.

Modification commands

Of course the examination commands would not be of much use unless you could subsequently change the values of memory locations or registers. These modification commands are obviously potentially dangerous, especially if you start to change the values of the control registers, like PC, SP or P. However they can be very useful in that they enable you to continue running the program after a point where it went wrong. It also allows you to put in test data, rather than having to write the input and output routines while you are still testing small modules of the program. The danger is that you can 'modify' any part of memory, including the Kernal (operating system) of your Commodore 64, or even the screen memory.

Error trapping with a debugger

When you are writing a machine code routine it is good practice to include breakpoints in the code at the time of writing, in the same way that you use breakpoints in BASIC to isolate important sections of a program. This means that the program will stop automatically at certain places.

This makes the testing process easier, as you can now check the intermediate results, and see whether things are developing as you intended them to. These breakpoints, and continuing the program flow afterwards, are directly analagous to the STOP, GET, GOTO and CONT controls which you have already used with BASIC programs.

The examine instructions perform the same function that you were when printing out the variables' values. The registers, and designated areas of memory, are performing exactly the same functions as the variables in a BASIC program; they are storing intermediate results for you.

Remember that when a critical error occurs in a machine code program it is often too late. Unlike BASIC, where you can usually escape from a program by pressing the RUN/STOP and RESTORE combination, you cannot always get out of a machine code program. And switching the Commodore off (and then on again) is a waste of your time.

When you do find an error, it is important to ascertain whether it is an error of coding or an error of logic. If the mistake is simply in the way the code has been programmed, then you can look at the piece of code, and modify it, editing as necessary. However, if the flaw is one of logic then it is a good idea to move away from the routine and to look at the sequence of events you are trying to program (see Chapter 11).

A logical flaw can result in you having to rewrite large sections of a routine, whereas coding mistakes only usually need one or two lines to be changed, to effect a correction.

Making good use of the editor/assembler

You should make the best possible use of the fact that you can edit assembly language programs using the editor. Use semi-colons to edit out calls to machine code subroutines in the same way that you use REMs in BASIC to selectively edit out sections of a program. This allows you to take advantage of the benefits of modular programming, and should reduce the amount of time you spend writing a machine code program. You can also add in descriptive comments without difficulty. This will make future debugging easier, as will the use of a program shell, as described in Chapter 11.

STORING MACHINE CODE AFTER ASSEMBLY

When a machine code program has been written and assembled into numbers in memory (and works) it is important to be able to retrieve it for future use. How you do this depends on the form in which you want to have the numbers. One method is to save the area of memory containing the routine as a block of bytes, using the appropriate command from your assembler. However, you may find it more convenient to store the assembled program in DATA statements in a BASIC program, where it can be examined, and easily loaded or saved.

If you want to do this then you should use a routine to print out the PEEKs of the area of memory where the machine code routine resides. These should then be printed across the screen, five to a line. This clear presentation allows you to copy the numbers accurately onto a piece of paper, again five to a row, or to print them out, should you have a printer.

When you come to type the same information back into DATA statements you should put five pieces of data to a line — this may not be as many as you can squeeze on, but it provides a more reliable method of checking the transfer. Remember that any error in either transfer (screen to paper, or paper to DATA statements) is fatal, and could well require the whole routine to be transferred again. And the process is tedious enough as it is, without having to do any more work than you need to.

If you make sure that you check the information one line at a time, in both cases, then you will find the method outlined here both efficient and reliable. There are better

methods of doing the transfer, the best of which is to PEEK the machine code routine and then POKE the results into dummy DATA statements. However, this requires a thorough understanding of the workings of the BASIC interpreter and I would advise sticking to a simple method unless you are prepared to dig deep into the heart of your Commodore 64.

To effect a transfer like this you must set up two BASIC programs in parallel, at different places in memory, switching the BASIC pointers when you want to SAVE the new DATA statements. The principal danger is that it is all too easy to lose one or both of the programs if you are altering the system pointers.

TROUBLESHOOTING IN MACHINE CODE

Most of the troubleshooting techniques which you have already used (with some success I hope) from previous chapters, are equally applicable to machine code. The most important of these are structure, documentation, modular development, selective testing, and careful checking of programs.

Structured programming

A structured program is almost always the result of logical planning. The idea of planning the steps which a program might take, before programming, is very important. This is true in BASIC, as you have probably found in this book, but it is even more so in machine code, as modifying and altering machine code programs is so time consuming. Whereas BASIC's line numbering system allows you to add or take away parts of a program simply, editing of machine code listings is more complex.

Modifications do not just cost effort, but time too. The lack of implicit line numbers means that changes to the program have to be made at the precise point where they are needed, rather than afterwards, or as an afterthought. Even then you may not be in the clear — it all depends on the assembler that you are using. Any time you spare for thoughts on the program structure before coding will be guaranteed to repay your investment in saved debugging time.

Documentation

In BASIC programs, although documentation is desirable, it is not essential. But if you are to understand a machine code

program (even if you wrote it and it is short), then documentation is crucial. Without comments in the program on what is happening, and what the various instructions and sections of the code are doing, you will soon get lost.

Although it is tempting and may seem quicker to write programs without comments, it does not work this way. The time saved at the outset will be far less than that you have to spend later, poring over the program and trying to work out just what you were trying to do. Of course as you get more practised you will not necessarily need a comment on every line, but it is better to be heavy-handed with machine code documentation. In BASIC you could always get away with no comments at all — in machine code you cannot.

Modular programming

The only way to structure a program effectively in machine code is to break it down into very small modules. This can be done in the same way that BASIC programs were split up into subroutines, as you saw earlier, though with machine code you have to take even smaller chunks of program into each subroutine.

In the same way that documentation repays your efforts, modularisation does too. With a program broken down like this you can proceed with testing and debugging in a logical way. This means that when you know you have a mistake, it can be quickly isolated to one small section of code; if you know that an error lies within one specific section then you are more likely to be able to find (and correct) it. The erroneous sections can be found by editing out other parts of the program.

Selective testing

The modular process also allows the separate development of each section of program. Once you are sure that each part works you can then string them all together; any resulting errors must logically be caused by the interaction of the various routines, as you know that they all work individually. This process of selective testing is the best way of working with machine code, unless of course you are the one programmer in a million who can type in ideas and watch them work straight away.

Careful checking

The last idea from BASIC troubleshooting which can be carried over to machine code is that of careful checking. It is good practice to look through the whole program at least

once before assembling it. This last look through often enables you to catch the obvious mistake (which will cause the program to fail), before it actually happens. At first you will find that this practice of checking does not seem to help, but after a time you will start to pick up bugs by scanning in this way.

When you examine the code, try to look at it with a fresh eye, and see if you can spot anything that does not look right. Often that is all there is to it — you spot a pattern which does not feel right, in the same way that a proof-reader spots the pattern of a misspelt word. You can then look more closely, and try to find the mistake. Although sceptical of this method when I first found out about it, it has stood me in good stead, and is an effective way of picking up errors.

GENERAL TROUBLESHOOTING TIPS

When you have found an error in a machine code program it can sometimes be corrected by simply poking in new hexadecimal values to replace the faulty ones. However, unless you are absolutely sure of what you are doing I would recommend that you go back into the source program in the editor/assembler, and reassemble the listing after making a correction.

For one thing, if the new instruction you have poked in is just one byte longer than the previous one, then you will have overwritten another part of the program. One byte shorter and you will have half of the previous instruction still left in memory. Even if the correction has worked you will still have an error in the source program.

With machine code, even more than with BASIC, it is a good idea to work with the motto: 'prevention is better than cure'. Time spent designing a program will always compensate for time you would otherwise spend in frustrating debugging (and nothing is more frustrating than debugging assembler or machine code). Even so, you are still likely to spend most of your time with machine code trying to debug it, rather than write it.

In an endeavour to let you spend less time debugging, here are some of the commonest problems which you may find with 6502 assembly language.

Addressing modes

One of the more complicated facets of any machine language is its addressing modes. Are you referring to the

contents of a register, the contents of the address in a register, or the contents of the contents of that address?

The 6502 has no less than thirteen addressing modes, and the matter is not helped for the beginner by the presence of the index registers X and Y, though these actually add a lot of power to the processor. There really is plenty of room for confusion, and it is possibly the most common machine code bug of them all.

Learning the addressing modes (and checking the program before assembly), is probably the only way that you can get around the problem, though the addition of comments is also sure to help. For example, if Y contains $20, the instruction:

```
LDA $1000,Y     ;Load A with contents of 1000+Y
```

gives a clue to the absolute indexed addressing mode being used. In this instance, after the instruction has been executed, register A will contain the contents of hexadecimal location 1020 (the contents of decimal 4128).

The trouble with comments

However, the addition of comments is not without its own problems. If you forget to put in the semi-colons you can end up with a disaster, as the assembler will try to understand your written comments as a series of instructions.

This is usually easily spotted, but, like most machine code errors, wastes time. The moral is that comments should be added, but check these as diligently as you check the rest of the program.

Using the stack

The stack can also cause problems; one of the main principles of machine code is that you can use the stack as a kind of temporary store, pushing variables (registers) onto it, and then popping them back off later on. This can also be used as a way of swapping register values, but the problem comes when registers are pushed on and popped off in the wrong order.

If you have PHA, PHP, PLA and PLP in succession then you will have swapped the values over, whereas you probably just wanted to save them off and then restore them. Always check that saving and restoring operations are performed in the opposite order to one another — unless you are deliberately swapping registers.

The other problem which can crop up with the stack is that as the SP register can only hold eight bits, you cannot have a stack bigger than 256 bytes of information. This means that it is not impossible to run out of space. This error can easily be fatal, and it is worth being on your guard against it.

Confusion between different machine languages

If you are lucky enough to know more than one machine language, then you could well find yourself in trouble, if you get confused about which one you are using now. PHA and PLA are one of the classic examples of this. Most languages use the word PUSH, but the 6502 specifies the register, and equally several use the word PULL or POP, while the 6502 specifies PLP or PLA.

In yet another case the word PULS is used instead. Other problems between languages include other differences in the instruction set, different addressing modes and registers, and their names — in other words just about every difference you would expect, despite the theory that all machine code languages are roughly the same.

Registers

Mixing up registers is also easy, and can be dangerous. If you are using one register to store a count and you test another by mistake, then this could cause a bug which is hard to trace later on, even if you go through the program step by step. Equally you can get problems with the registers if you forget that they have a fairly small size — one byte is standard, and only numbers between −128 and +127 can be used. Anything larger causes an overflow, unless you are using solely positive numbers, in which case the range is 0–255.

Registers are the key to machine code, and remembering which registers are doing what at any one time is particularly important on the 6510. Instructions like the register transfers TAX and TAY use the accumulator implicitly, and you could forget that A will be wiped out by the instruction. It obviously pays to be careful.

Flags

As well as registers causing these problems, you can also run into trouble if you test the wrong flags. The flags are stored in the P or status register, and are tested implicitly by most of the branch instructions. If the wrong flag is examined then disaster is just around the corner.

Constants

The same can be said of almost any occurrence of constants. These come in several forms, from physical addresses, to names equated to constants at a program start, to numbers you want to use as counters, but they all have one problem. It is very important to remember what base you are using with a number. It is vital to see the difference between hexadecimal 100 (256), octal 100 (64), binary 100 (8) and decimal 100 (100) — for obvious reasons. Fortunately hexadecimal numbers are universally prefixed on the Commodore by the dollar sign ($).

Labels and assembler fields

Other common problems are those of using the same label accidentally in two places (this is especially likely to happen in longer programs) and confusing the assembler fields (label, instruction and comment). With the latter it is a good idea to lay out the program with respect to these fields, as the program is then more readable, and hence more comprehensible.

These are just some of the problem areas of machine code, but you will find that others can be spotted too, if you use the same troubleshooting techniques which have been described here and for BASIC.

CALLING MACHINE CODE FROM BASIC

The key to calling up the machine code routines once they have been assembled is the BASIC keyword SYS. This tells the Commodore 64 that the next number is the start of a machine code routine, and directs the processor to start executing machine code from that location. After the machine code routine has finished an RTS instruction will make control return to the line in the BASIC program following the SYS command. Note that you are very unlikely to have anything other than disaster if you use SYS at random — the chance of success is minute.

You can pass information to your machine code routine from BASIC by POKEing it into the variable locations in your routine, though obviously this should be done with great care as you could easily alter an instruction rather than a variable.

RELOCATABLE MACHINE CODE PROGRAMS

As you know it is an advantage to make machine code programs relocatable. In theory this is not difficult, since it

only requires you to set the origin and make everything relative to this point.

However, the longer it gets, the more difficult and the more inefficient a relocatable routine becomes. In practice the most complicated routines are not in fact relocatable — they are too long.

Before attempting to relocate a machine code routine be as sure as you can to be certain that you can do so without causing the routine to crash. In practice this is difficult, because you can put a machine code routine wherever you want to in the 64K of RAM, even though many areas of it will cause problems. Consulting the memory map should help you avoid the most obvious pitfalls — like putting a machine code routine on top of the zero page or Kernal. Short programs should always be written so that they are relocatable.

Appendix A
THE COMMODORE 64 COMPUTER

This Appendix looks at the advantages and disadvantages of the Commodore 64 and compares its features with other home computers.

Full-travel keyboard The Commodore 64 has a full-travel keyboard which allows you to type faster and more accurately than is possible on the membrane type such as that used on a Spectrum. This is particularly advantageous both when programming and word processing.

Keywords On the Commodore 64 all keywords must be typed in full and are subsequently reduced to tokens by the BASIC interpreter. The practical result of this is that you cannot type a keyword with a single keypress, as you can on the Spectrum, for example. The advantage of the Commodore method is that you do not have to search for the keyword you want on the keyboard — you simply type the word instead. Note that you can also use abbreviated forms of most keywords and that once you get used to the way the Commodore 64 is programmed you can save time by doing this. See Chapter 2 for more details on typing in programs.

Loading and saving programs Loading and saving programs on the Commodore 64 is not difficult, though there are one or two things to remember. The first is that since the Commodore cassette unit has no volume control you have no choice over what level the volume is set to — which leaves one less thing to worry about. Loading and saving are achieved with the simple LOAD and SAVE commands and filenames must be enclosed in quotes. When the Commodore is loading or saving the screen is often switched off (for reasons detailed in Chapter 7), and you will therefore not have any indication of what is happening! The last thing to remember is that as the Commodore 64 has a remote lead to the cassette recorder, some of the switching on and off of the recorder can be left to the computer. Further information on loading and saving can be found in Chapters 5 and 6 respectively.

Control keys There are few keys used for control purposes on the Commodore 64, but the most important is the RUN/STOP and RESTORE sequence which will break out of most programs running on your computer. Other control sequences can be used to change mode — pressing the Commodore key and CTRL keys together will change the character sets in memory between the two available.

The cursor keys on the Commodore 64 are at the bottom right-hand corner of the keyboard and are used in conjunction with SHIFT to give the four directions. These keys and the INST/DEL key are the ones used most while editing. Remember that editing on the Commodore 64 can take place anywhere on the screen — like the BBC Micro but not the Spectrum. See Chapter 2 for more on using the keyboard.

Input from the keyboard Input from a user while a program is running is different on most computers, but is simple on the Commodore 64 which uses the standard BASIC command INPUT for most applications and GET for accepting single character presses. One thing to note is that GET will work as soon as it is called, so to accept the next keypress GET must be placed in a loop. Also remember that GET takes the next character from the keyboard buffer — the implication of this being that the buffer should be cleared first if you want to get a keypress directly from the keyboard. Chapter 5 details more about all input, and keyboard input in particular.

Disk drives The Commodore 64 can have disk drives attached to it, but the disk operating system is inside the drive rather than the computer itself — unlike the BBC Micro, for example. On Commodore disk drives you can use most of the same commands which you would use for cassette, but these must be sent to the drive, rather than the cassette by adding **,8** to the commands.

In addition, the disk drive itself contains extra commands in a separate ROM — these can only be used with the drive. Before going too far with disk drives it is advisable to have a good look at the manual, after which the disk drive information in Chapters 5 and 6 in this book should help sort out any remaining problems.

Direct merging On the Commodore 64 you cannot directly merge BASIC programs together. However, utilities which allow you to do this can be obtained both commercially and privately, and they will make any programmer's life much easier. See Chapters 5 and 6 on input and output, and Chapter 10 on memory management for more details on the way that BASIC is stored in memory.

Storing and handling of machine code There is no built-in machine code editor or assembler on the Commodore 64, but these are easily available commercially, one of the most useful and popular being Commodore's own plug-in cartridge. This will create machine code files in memory, and allow you to save these to tape or disk. Machine code can also be stored as a list of numbers in DATA statements which are then poked by the BASIC program into the correct position in memory. Many people find this an easier way to handle machine code as the numbers can actually be seen and read. See Chapter 11 for more details on how to use machine code, and Chapter 5 for details of saving.

Joystick and printer interfaces All computers have some interfaces built in, but few have all those possible. The Commodore 64 has joystick interfaces connected up to a special controlling chip (see Chapter 5), but has no external printer interface. Of course you can link Commodore printers directly to your 64 as they have the necessary interfacing built in, but different makes of printer will need an extra interface. The commonest two available are the Centronics and RS-232 standards, and the one you choose will probably have more to do with the printer you buy than the interface standard. The RS-232 has an added advantage in that the same interface can sometimes be used for communications as well as for printing. The drawback with an RS-232 is that it is much slower than a Centronics interface, which is a parallel interface and therefore much faster than the serial RS-232.

Renumber routines The Commodore 64 has no built-in renumbering routine, so it is a good idea to leave plenty of space between the lines so that others can be added later (see Chapter 11). Renumbering routines are available both commercially and from magazines and clubs, and it is well worth finding one. However, before using a renumbering routine check that it works properly, and includes changing the numbers in GOTO and GOSUB statements.

Trace One feature some micros offer is that of being able to run a trace on a BASIC program so that you can see what is happening as the program executes. This feature is not available on the Commodore 64, but there is software which will do this for you. Although something of a luxury in BASIC, a trace is almost essential when using machine code (see Chapter 12).

Screen and graphics editors If you are writing games software you will often want to be able to design a whole screen. Because there are no simple drawing commands on the Commodore 64 this can be very time consuming, even if you use the block graphics available. However, you can buy software which will allow you to design and save screens. This will be much faster and more efficient than programming every line yourself, though commercial editors can be expensive. See Chapter 7 for further details of graphics.

Word processing One reason for buying a micro is word processing, but since there is none built into the Commodore 64 it means that you will have to buy a package. There are several available, and though they are all limited by the memory size of the computer, a disk-based package can give you very effective word processing at a remarkably low cost. Look at any package critically and test it fully before buying.

The BASIC One area where the Commodore 64 is different from other machines is in the language it uses. Although it is a version of BASIC, it is not quite the same as other BASICs. For example, there is no IF . . . THEN . . . ELSE statement, and no WHILE or REPEAT loops, though there is an ON . . . GOTO statement. However, it does have excellent graphics and sound capabilities although these have to be accessed via PEEK and POKE rather than through extra BASIC keywords. Commodore BASIC is generally considered to be rather poor compared to other versions (BBC BASIC, for example), though the machine is powerful, once you know how to use it. Chapter 4 details all of the differences and the particular problem areas you might find with the BASIC, while the subsequent chapters look at the specific areas of BASIC used for special functions like input, output, graphics and sound.

Other differences

Naturally there are also many other differences between the Commodore 64 and other computers, the most obvious ones being in the way that the screen is used, the colours available, the graphics and sound facilities and the way that memory is handled, but these topics are all covered in detail in the relevant chapters. Suffice it to say that although some parts of the Commodore 64 are similar to other computers, many are not.

Appendix B
COMMODORE ERROR MESSAGES

This is a complete list of the error messages which can be generated by the Commodore, along with an indication of the likely cause of each.

BAD DATA If you are using files and the program is expecting numeric data but actually receives string data, then this error message is generated. The easiest way of avoiding the problem is to accept all input data from files as strings and then convert it in the program to numerics. *(Chapter 5)*

BAD SUBSCRIPT If you use a DIM statement to set up an array then any subscripts you use later in the program must be within the limit specified in the DIM statement. This error is the result of trying to access an array element which does not exist because it has not been declared. *(Chapter 4)*

BREAK When you hit the RUN/STOP key and the program stops then this message is generated — in fact it is not really an error but it shows you how and why the program stopped. *(Chapter 3)*

CAN'T CONTINUE This message is generated after certain editing commands, or if a program has never been run or has crashed. You cannot, then, continue running from where the program last stopped. (*Chapters 2 and 3)*

DEVICE NOT PRESENT When using device handling commands like CMD, OPEN and CLOSE you must specify the device required by number. If the peripheral is not attached or is not switched on then this error message will appear. *(Chapters 5 and 6)*

DIVISION BY ZERO Since dividing any number by zero yields an infinite result it is not allowed on your Commodore 64 — an attempt to divide by zero will provoke this message. *(Chapter 4)*

EXTRA IGNORED When the INPUT statement is used you can type in as many items separated by commas as you want. However, anything surplus to the requirements of the INPUT statement will be ignored, as well as generating this message. *(Chapter 5)*

FILE NOT FOUND This error can happen in two cases: the first when a file is being searched for on tape and the end of tape marker is found, and the second when a file searched for on disk does not exist (ie it is not present in the directory). *(Chapter 5)*

FILE NOT OPEN Any of the file handling commands can cause this error if the file specified in a command has not first been opened. *(Chapters 5 and 6)*

FILE OPEN If you try to open a file which is already open then you will cause this error message to be generated. *(Chapters 5 and 6)*

FORMULA TOO COMPLEX You will sometimes find that the computer cannot handle the number of brackets being used in an arithmetic expression, or the number of separate arguments in a string expression. The solution is to split the expression up into two simpler ones and evaluate them separately, combining the results afterwards. *(Chapter 4)*

ILLEGAL DIRECT The INPUT statement cannot be used in direct mode, but only in a program. Any attempt to use it as a direct command will cause this error. *(Chapter 5)*

ILLEGAL QUANTITY If any part of an expression, or the result of an expression is out of the allowable range then this error is generated. The commonest occasion is when you try to poke a number larger than 255 into a memory location.

LOAD If some problem is encountered when loading from tape then this is the message which appears. *(Chapter 5)*

NEXT WITHOUT FOR If a NEXT is encountered which cannot be matched with a preceding FOR then this message is generated. *(Chapter 4)*

NOT INPUT FILE If a file has been specified as output only and you try to input from it, then this message will be generated. *(Chapter 5)*

NOT OUTPUT FILE This error will be generated if you have specified a file as input only and then try to output to it. *(Chapter 6)*

OUT OF DATA If a READ statement is trying to read DATA but finds that there is no more left, then this error will be generated. *(Chapter 5)*

OUT OF MEMORY When there is no more RAM available for program variables or arrays then this error will occur, but it can also be generated by too many nested FOR loops or a recursive subroutine. *(Chapters 4 and 10)*

OVERFLOW A calculated result is larger than the Commodore 64 can handle — any number over 1.7×10^{38} will cause the error.

REDIM'D ARRAY You cannot dimension the same array with a DIM statement more than once. Should you attempt to do so then you will get this message. *(Chapter 4)*

REDO FROM START When you use the INPUT statement to accept data from the keyboard, then typing *character* data when a *number* is expected will cause this error. Notice that it does not crash the computer but that you simply retype the offending piece of data and continue on your way. *(Chapter 5)*

RETURN WITHOUT GOSUB This message occurs if a RETURN statement is encountered by the program and the computer cannot find an associated GOSUB which called the subroutine. This commonly happens because GOTO has been entered instead of GOSUB. *(Chapter 4)*

STRING TOO LONG Strings on the Commodore 64 can be anything up to 255 characters long. Anything over this will cause the error message above. *(Chapter 4)*

?SYNTAX ERROR When you are typing in programs on the Commodore 64, the computer does a certain amount of checking for you, and if it cannot recognize a keyword, or if there is the wrong number of brackets or incorrect punctuation then this error will be generated. *(Chapter 2)*

TYPE MISMATCH Data in the Commodore 64 is of different types (strings or numerics). If one type is used when the other type is expected, then the mismatch error is generated. A typical example would be to try and take the CHR$ of a word rather than a number. *(Chapter 4)*

UNDEF'D FUNCTION If you try to reference or call a user defined function which has not in fact been defined (using DEF FN) then this error message will be generated. *(Chapter 4)*

UNDEF'D STATEMENT When you use the RUN, GOTO or GOSUB commands the first thing the Commodore does is to see if that line number actually exists. If for some reason it does not, then this error is generated. Note that this is different from computers like the Spectrum which simply take the next nearest line number if the specified one does not exist. *(Chapter 4)*

VERIFY If you verify a program on disk or tape then it is compared with the current program in memory. If they are not identical, byte for byte, then this error is generated. *(Chapter 6)*

INDEX

Most home computer owners spend more time trying to cure the bugs in their programs than writing them in the first place. This book, by a professional programmer, shows you in clear and logical steps how to cut down on your debugging time. He shows you how to lay out a program clearly, how to write it in modules, each one of which can be tested before being incorporated into the whole. He looks at the most common things which go wrong in the areas of sound, graphics and subroutines. Above all, he stresses the importance of making regular saves and backups of your work.

Whether you're a beginner taking your first steps in BASIC or already tackling machine-code programs you are bound to find this book useful.

Further machine-specific versions of this book are available for the Spectrum and the BBC/Electron.

ISBN 0-7126-0600-9

£7.95

ISBN 0 7126 0600 9

Century Communications Ltd

9 780712 606004